
CHAPTER

11

PROCESSOR FAMILIES

CHAPTER OBJECTIVES

In this chapter you will learn about:

- The ARM, PowerPC, Sun SPARC, Compaq Alpha, and Intel IA-64 processor families which implement RISC ISAs
- The Motorola 680X0/ColdFire and Intel IA-32 processor families which implement CISC ISAs
- The Hewlett-Packard HP3000, whose instructions are designed to operate on a stack data structure

Chapter 2 showed how basic programming concepts are implemented at the assembly language level and illustrated the need for various machine instructions and addressing modes. In Chapter 3, the ARM, Motorola 68000, and Intel IA-32 instruction set architectures were used as examples. In this chapter, we continue the discussion of those three ISAs, elaborating on features found in the various members of the processor families that implement the ISAs, spanning a range of cost and performance levels. The ARM architecture exhibits RISC characteristics, and the Motorola and Intel IA-32 architectures have CISC features. The RISC-style PowerPC architecture, whose processor implementations compete in the same target market as IA-32 processors, is described. We also discuss the Sun SPARC, Compaq Alpha, and Intel IA-64 RISC architectures, which have 64-bit address and data lengths. These 64-bit processors are intended for use in high-performance workstations and servers. Finally, a very different approach, which is now mainly of historical interest, that of the Hewlett-Packard HP3000, shows how an instruction set can be organized for a machine that has computation facilities heavily influenced by the use of a stack data structure for holding operands.

Chapters 2, 3, 7, and 8 described various RISC and CISC approaches to processor design. We now briefly review some of the salient features of each approach before giving the family examples that exhibit these properties. CISC instruction sets provide many powerful instructions intended for more direct implementation of high-level language operations and program sequencing control structures. However, the execution of such instructions can become quite complex. The rationale of the CISC approach is that it leads to fewer machine language instructions for a given high-level language program, thus leading to shorter program execution times. This is true if complex instructions can be executed quickly and efficiently. In practice, however, the implementation of these instructions with the goal of meeting high instruction execution rates has proven to be very challenging, and has required relatively large chip area. In addition, CISC instruction sets are difficult targets for optimizing compilers.

The RISC approach of using relatively simple instructions may at first appear to be less effective than the CISC alternative because more RISC instructions are needed to perform a given computational task. However, RISC instructions are well-suited to pipelined execution that leads to high execution rates. A key advantage of RISC instruction sets is that they can be used effectively by optimizing compilers. Another advantage is related to *Very Large Scale Integration* (VLSI) fabrication technology. Because of the smaller chip area needed for instruction handling and sequencing control in RISC processors, more space is available for larger register sets and larger on-chip caches. Higher performance results because off-chip data and instruction accesses are reduced.

Taking all factors into account, the result is that although both approaches have led to very competitive commercial products, new machines developed since the early 1990s have predominantly featured RISC-style ISAs.

We also must emphasize, as explained in earlier chapters, that many factors other than the style of the instruction set are very important in the design of a processor that provides a high rate of instruction execution. Multiple, pipelined, functional units that enable superscalar execution of programs, as described in Chapter 8, are essential. Also, optimizing compilers that generate efficient machine language code from high-level language programs must be developed along with the hardware.

11.1 THE ARM FAMILY

In Part I of Chapter 3, we described the ARM instruction set architecture as an example of the RISC style of instruction set design. ARM processors are mainly intended for embedded system applications. Most implementations must therefore meet low-cost and low-power requirements. In many cases, such as mobile telephones, the target device is battery powered, and voltages in the range of 1 to 3 volts are used. Compared to the high-performance Intel Pentium processors that are targeted for the PC market, low-end ARM processors are significantly less complex, and require only a fraction of the transistors needed for Pentium chips. We will examine some of these issues in discussing various implementations of the ARM ISA. The book by Furber [1] contains a wealth of information on all aspects of ARM history, architecture, and implementation. The articles in [2] and the ARM web site [3] also provide detailed information.

From its original development in the mid-1980s, through to the year 2000, there have been five major versions of the ARM ISA, labeled v1 through v5. Version v3 is described in Chapter 3, and the ARM7 processor, developed in the mid-1990s, implements this version. A later section will describe processor implementations and their properties.

Relative to version v3, the other versions are briefly described as follows. Versions v1 and v2 supported only 26-bit memory addressing, and the 32-bit product multiply instructions were added in going from v1 to v2. Version v3 introduced full 32-bit addressing for byte and 32-bit word operands. Version v4 contains full 64-bit product multiply instructions, as well as the 32-bit product instructions. It also adds load and store instructions for 16-bit (half-word) data operands. Version v5, and an extension of it labeled v5E, add specialized instructions for: managing software breakpoints for debugging, normalizing numbers for the software implementation of floating-point arithmetic operations, and performing addition and multiplication operations on 16-bit operands to allow efficient execution of digital signal processing routines.

In addition to this evolution of the standard ARM ISA through five versions, in which all instructions are encoded into a 32-bit word format, a more compactly encoded subset coexists with versions v4 and v5. It is described in the next section.

11.1.1 THE THUMB INSTRUCTION SET

The ARM ISA specification includes a compact encoding of a subset of the v4 and v5 versions of the full set of instructions. The subset is called the *Thumb* instruction set, and the version names are extended to v4T and v5T to denote this inclusion. All Thumb instructions are encoded into a 16-bit half-word format.

The practical motivation for the Thumb instructions is that they lead to a reduction in memory space needed to store programs used in low-cost and low-power embedded system applications. The ARM7TDMI processor implements version v4T of the architecture. This processor, along with a small amount of RAM memory and digital signal processing hardware, fabricated on a single chip, is suitable for the mobile telephone application.

Programs consisting of Thumb instructions are executed as follows. The instructions are fetched from memory and dynamically (that is, as they are to be executed) “decompressed” from their highly encoded 16-bit format into corresponding standard 32-bit ARM instructions, and then executed. This is how they are handled in the main stream low-end processors. In some high-performance processors, the Thumb instructions are decoded directly for execution, avoiding the decompression step into standard ARM instructions. A bit in the Current Program Status Register (CPSR), labeled T, determines whether the incoming instruction stream consists of Thumb ($T = 1$) or standard 32-bit ARM instructions ($T = 0$). An application can contain a mix of Thumb and standard instruction routines.

Two of the main differences between Thumb and standard instructions are: A number of Thumb instructions use a 2-operand format in which the destination register is also one of the source operand registers; and Conditional execution, which applies to all standard ARM instructions, is used mainly for branches in the Thumb set. These changes clearly lead to savings in instruction encoding bit space.

11.1.2 PROCESSOR AND CPU CORES

The ARM company produces and licenses designs for ARM processors and closely associated components such as caches and memory management units. Companies that produce embedded systems and other application-specific computer products acquire these designs for incorporation into their products. In most cases, the ARM designs are integrated with application-specific hardware on the same chip. To capture this intended usage, the ARM designs are called *cores*. Designs are provided by ARM in two different forms: *hard macrocell* or *synthesizable*. The hard macrocell version is a detailed physical layout, targeted to a particular chip fabrication process. The synthesizable form is a high-level language software module that can be synthesized using a suitable cell library in the required target technology. This form allows a number of options on processor functionality to be easily included or omitted. The ARM7TDMI processor is a hard macrocell core, and the ARM7TDMI-S is its synthesizable version.

ARM designs are classified as either processor cores or CPU cores. A processor core contains only a processor and associated address and data bus connections. A CPU core contains cache and memory management components in addition to a processor. The name CPU is somewhat misleading, because traditionally it has meant Central Processing Unit, but we use the name here because it is an identifiable ARM term. We now give brief descriptions of some representative processor and CPU cores.

ARM7TDMI Processor Core

This core is commonly used for low-cost low-power applications. The processor has a simple 3-stage pipeline consisting of fetch, decode, and execute stages. It realizes version v4T of the architecture, supporting both the Thumb and standard instruction sets. Typical operational parameters are a 3.3 V power supply and a 66 MHz clock rate. But this core design can be synthesized for 0.9 V operation for low-power battery supply applications or for over 100-MHz clock rates to achieve higher performance.

ARM9TDMI and ARM10TDMI Processor Cores

These two processor cores are based on 5-stage and 6-stage pipelines, respectively. They have separate instruction and data ports to provide much higher performance levels than the ARM7TDMI can provide. At clock rates of 200 MHz and 300 MHz for these processors, performance levels of the 7, 9, and 10 versions of this processor core are in the ratios 1:2:4. The ARM10TDMI has a wider 64-bit path to each memory port, as compared to 32-bit paths for the other two processors. The ARM9TDMI and ARM10TDMI implement versions v4T and v5TE of the ISA, respectively. Both of them decode Thumb instructions directly for execution. Cache memories must be used with both of these cores to achieve the higher performance levels.

ARM720T CPU Core

This core consists of the ARM7TDMI processor core combined with an 8K-byte unified instruction and data cache, and virtual memory management hardware. The cache structure has 16-byte blocks and is 4-way set associative. The memory management unit uses a 64-entry associative translation lookaside buffer for holding recent translations. The clock rate for this integrated unit can be up to 60 MHz. The added cache and MMU circuitry increases the total silicon area required by a factor of about 5 over that required by the processor core alone, and power consumption triples.

ARM920T and ARM1020E CPU Cores

These CPU cores, based on the ARM9TDMI and ARM10TDMI processor cores, have separate instruction and data caches. Each of the caches in the ARM920T contains 16K bytes and has 32-byte blocks, with 64-way set associativity. There is an MMU for each memory port, and each of them has a 64-entry associative TLB. The ARM1020E has 32K bytes in each cache; otherwise, the caches and the MMUs are similar.

StrongARM SA-110 CPU Core

The StrongARM CPU core was developed by ARM in collaboration with Digital Equipment Corporation (now folded into the Compaq company), and the SA-110 version is manufactured by Intel. The processor component implements version v4 of the architecture. It does not support the Thumb instruction set. Otherwise, the processor is comparable to the ARM9TDMI processor core. The performance of the StrongARM SA-110 is comparable to that of the ARM920T, but it is implemented using an earlier technology and has higher power consumption at a 200-MHz clock rate.

The StrongARM processor has a 5-stage pipeline. There are separate 16K-byte instruction and data caches. Each cache has 32-byte blocks and 32-way set associativity. The translation lookaside buffers for each of the caches have 32 entries. The high-speed multiplier circuitry has a latency of three or fewer clock cycles, designed for good performance in digital signal processing applications.

11.2 THE MOTOROLA 680X0 AND COLDFIRE FAMILIES

In Part II of Chapter 3, we described the Motorola 68000 processor. Here, we discuss the key features of the follow-on processors in the 680X0 family and the closely associated ColdFire family. The book by Tabak [4] and the Motorola web site [5] describe these processors. Some general comments on Motorola processors are given first.

The 68000 processor was introduced in 1979. Through the 1980s and early 1990s, the 68000, 68020, 68030, and 68040 were targeted for the personal computer market, and were used in Apple computers. The latest member of the 680X0 family is the 68060, introduced in the mid-1990s. The 68060, and the closely related ColdFire family, are targeted for the embedded system market.

11.2.1 68020 PROCESSOR

The 68020 is much more powerful than the 68000, mainly because of some significant architectural enhancements. These advances were made possible by improved VLSI technology and larger packages that removed many constraints of pin limitations. The discussion here on the 68020 also applies to the 68030 and 68040. Later, we describe the additional enhancements found in the 68030 and 68040.

The 68020 has external connections for 32-bit addresses and 32-bit data. Although its data bus is 32 bits wide, the 68020 can deal efficiently with devices that transfer 8, 16, or 32 bits at a time. The processor can adjust dynamically to the data bus width requirements of a particular device in a manner that is transparent to the programmer. The 68020 bus includes control lines that are activated by the devices connected to the bus to indicate the required size of their data transfers. Thus, the processor can deal with devices of different data transfer sizes without knowing the actual size before a data transfer is initiated.

The 68000 restriction that word operands must be aligned on even address boundaries has been eliminated in the 68020; operands of any size may start at any address. This means that 16- and 32-bit operands can occupy parts of two adjacent 32-bit locations in the main memory. Two access cycles are therefore needed to reach such operands, and this affects performance. The processor automatically performs these two accesses. From the address, the processor knows which 32-bit locations must be accessed and in what pattern the individual bytes from these locations should be assembled to obtain the desired operand.

Register Set and Data Types

Like the 68000, the 68020 has user and supervisor modes of operation. In the user mode, the registers available are essentially the same as those given in Figure 3.18 for the 68000. In the supervisor mode, however, the 68020 has several additional control registers intended to simplify implementation of operating system software.

The 68000 addressable data units are bit, byte, word, long word, and packed binary-coded decimal (BCD). In addition to these, the 68020 allows quad word, unpacked BCD, and bit-field data types. A quad word consists of 64 bits, and unpacked BCD has one BCD digit per byte. A bit field consists of a variable number of bits in a 32-bit long

word, and it is specified by the location of its leftmost bit and the number of bits in the field.

Addressing Modes

All 68000 addressing modes, shown in Table 3.2, are available in the 68020. Several extra versions of the indexed mode have been added to the 68020 to allow flexible and efficient access to data and address list structures.

The full indexed mode is more powerful because it allows a range of displacements, or offsets, and provides for a scaling factor. Recall that the 68000 syntax for the full indexed mode is

$$\text{disp}(An.Rk.size)$$

where the displacement is a signed, 8-bit number and the size designation indicates whether 32 or 16 bits of the Rk register are to be used in computing the effective address. The 68020 version of this mode allows the displacement to be an 8-, 16-, or 32-bit value. It also introduces a scale factor by which the contents of Rk are multiplied. The value of the scale factor may be 1, 2, 4, or 8. The syntax for the mode is

$$(\text{disp}, An.Rk.size * \text{scale})$$

Note that the displacement is given within the parentheses in this case. The effective address, EA, is computed as

$$EA = \text{disp} + [An] + ([Rk] \times \text{scale})$$

This mode is useful when dealing with lists of items that are 1, 2, 4, or 8 bytes long. If the scale factor is chosen so that it equals the size of the items, then successive items in the list can be accessed by incrementing the contents of Rk by 1.

Another powerful extension of indexed addressing is the memory indirect indexed modes, in which an address operand is obtained indirectly from the main memory. Two such modes exist. In *memory indirect postindexed* mode, an address is fetched from the memory before the normal indexing process takes place. Its syntax is

$$([\text{basedisp}, An].Rk.size * \text{scale}, \text{outdisp})$$

and the effective address is computed as

$$EA = [\text{basedisp} + [An]] + ([Rk] \times \text{scale}) + \text{outdisp}$$

Note that two displacements are used. A base displacement of 16 or 32 bits is used to modify the address in An , which is then used to fetch the address operand from the memory. This allows an address to be selected from a list of addresses stored in memory starting at the location given by the contents of An . The second displacement is the normal displacement used in indexed addressing, called outer displacement to distinguish it from the base displacement.

The second version is the *memory indirect preindexed* mode, in which most of the indexing modification is done before the address operand is fetched. The syntax for this mode is

$$([\text{basedisp}, An.Rk.size * \text{scale}], \text{outdisp})$$

and the effective address is determined as

$$EA = [\text{basedisp} + [An] + ([Rk] \times \text{scale})] + \text{outdisp}$$

In both of these modes, the values An , Rk , basedisp , and outdisp are optional and are not included in the computation of the effective address unless specified by the user. These addressing modes are useful for dealing with lists in which contiguous memory locations are used to store addresses of data items, rather than the data items themselves. The latter can be anywhere in memory.

A relative version of all indexed modes is available in which the program counter is used in place of the address register, An .

Instruction Set

All 68000 instructions are available in the 68020. Some have extra flexibility. For example, branch instructions can have 32-bit displacements, and several instructions have the option of using longer operands. Some new instructions are also provided, such as instructions that deal with bit-field operands.

On-Chip Cache

The 68020 chip includes a small instruction cache that has 256 bytes organized as 64 long-word blocks. A direct-mapping scheme is used when loading new words into the cache.

11.2.2 ENHANCEMENTS IN 68030 AND 68040 PROCESSORS

The 68030 differs from the 68020 in two significant ways. In addition to the instruction cache, the 68030 has another cache of the same size for data. The data cache organization has 16 blocks of 4 long words each. The 68030 also contains a memory management unit (MMU).

The execution unit in the 68030 generates virtual addresses. The cache access circuitry determines if the desired operand is in the cache, based on virtual addresses. The MMU translates the virtual address into a physical address in parallel with the cache access so that, in the case of a cache access miss, the physical address needed to access the operand in the main memory is immediately available.

The 68040 includes a floating-point unit that implements the IEEE floating-point standard described in Chapter 6. Instruction and data caches are included, as in the 68030. Memory management is improved over that in the 68030; the 68040 has two independent address translation caches that permit simultaneous translation of addresses for both instructions and data. The 68040 has a pipelined structure that permits fetching of instructions while previous instructions are still being processed. Two internal buses are used to transfer instructions and data from the respective caches. These buses, in conjunction with the two address translation circuits, allow simultaneous access to instruction and data caches.

Finally, the 68040 includes circuits that monitor activity on the external bus. This feature makes the 68040 suitable for use in multiprocessor systems. One of the key requirements in such systems is to maintain consistency of the common data that may

temporarily reside in several caches of different processors. The bus-monitoring circuits detect bus transfers that change cached data, as we describe in Chapter 12.

11.2.3 68060 PROCESSOR

The latest member of the 680X0 family is the 68060 [6], introduced in the mid-1990s, with clock rates ranging from 50 to 75 MHz. This processor is intended for the embedded system market. New organizational and fabrication features result in performance that is 2.5 times that of a 40 MHz 68040.

The 68060 is a pipelined superscalar processor. The pipeline has four basic stages, with an additional two stages if a memory writeback operation is required. Up to three instructions can be initiated per clock cycle. Three function units — two integer units and a floating-point unit — comprise the main instruction processing hardware. There are separate, on-chip, 8K-byte instruction and data caches. Each cache is 4-way set associative and uses 16-byte blocks. Two 64-entry, 4-way, set associative, translation lookaside buffers to facilitate virtual to physical address translation are provided with the caches. Dynamic branch prediction is used to enhance smooth flow of instructions through the pipeline.

11.2.4 THE COLDFIRE FAMILY

Since the mid-1990s, Motorola has produced a series of processor components and small computer configurations, called the ColdFire family, that is targeted for the embedded system market. The processors are based on the 68060 processor core. A number of different products are configured with small amounts of memory and I/O port hardware for parallel and serial connections. These products meet a range of power and performance requirements for different applications. Both hardware chip products and synthesizable software designs are available in the ColdFire family.

11.3 THE INTEL IA-32 FAMILY

Intel processors [7] have attained strong commercial success as evidenced by their wide use in notebook and personal computers. In the 1980s, Intel produced the first series of processors used in the IBM PC. They were based on the 8086 processor, introduced in 1979, which generated 20-bit addresses externally and handled 16-bit data internally and externally. (It is interesting to note that an 8-bit version of the 8086, labeled the 8088, was actually used in the first IBM PC to keep cost as low as possible.) Because the 8086 was encapsulated in a 40-pin package, the address and data transfers were time-multiplexed on the same set of chip pins.

Progressively more powerful processors that implement an evolution of the same basic instruction set architecture have been introduced. These are the 80286, 80386, 80486 [4], and the current Pentium series. The 80286 was a 16-bit processor. The others

all handle data and addresses both internally and externally in 32-bit sizes. The 80386 is the first processor in the IA-32 family. The 32-bit chips come in larger packages that obviate the need for address and data line multiplexing.

11.3.1 IA-32 MEMORY SEGMENTATION

In Section 3.16.1, we briefly discussed the use of segment registers (see Figure 3.37) in generating memory addresses in the IA-32 architecture. Here, we expand on that description. First, it is instructive to point out how the segment registers were originally used in the 8086. Current IA-32 processors can be put into a state that operates that way, called *real mode*. This allows current IA-32 processors to run 8086 machine code programs.

Real Mode

This address generation mode, used by the 8086 processor, views the memory as being organized in segments of 64K bytes each. A 64K-byte memory segment is spanned by the 16-bit effective addresses generated internally by the 8086 addressing modes. The processor uses the CS, SS, DS, and ES segment registers for accessing code, stack, and two data segments, respectively. The other two segment registers (FS and GS) were added in the 80386.

The 20-bit external memory addresses are generated as shown in Figure 11.1. The 16-bit value in a segment register is shifted left four bit positions to form a 20-bit address, which is the starting address of a segment. The 16-bit effective address generated by an 8086 addressing mode, labeled the offset in the figure, is added to the segment starting address to produce the desired 20-bit memory address.

Segments are located in different areas of the 20-bit address space by loading the high-order 16 bits of their starting address into the appropriate segment register. A total of sixteen, 64K-byte, nonoverlapping segments can be accommodated in the 1M-byte memory space spanned by 20-bit addresses. Note that segments can overlap. This is useful for sharing instruction and data space among different programs. The CS and SS segment usage is automatically determined for instruction and stack access references. The default for data access is to use the DS segment register. A prefix byte code can be added to an instruction to use the ES register for data accesses.

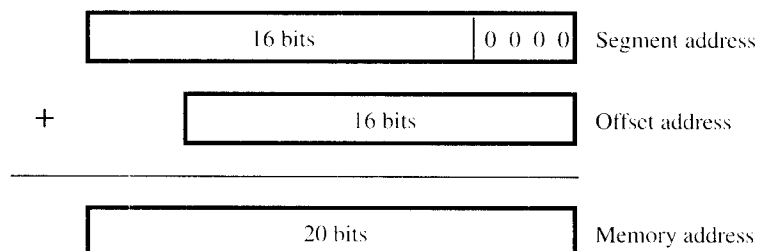


Figure 11.1 Generation of a memory address in the 8086 processor.

Protected Mode

IA-32 architecture processors normally use the *protected* mode for generating memory addresses. Figure 11.2 shows the most general way to generate a physical address using the contents of base and index registers along with a displacement value contained in the instruction. A 32-bit effective address is determined by multiplying the contents of the index register by a scale factor of 1, 2, 4, or 8 and then adding the result to the contents of the base register and the displacement. The high-order 14 bits of one of the six segment registers (shown in Figure 3.37) specifies a *selector*; which is then used as an index into a segment descriptor table from which a 32-bit base address is obtained. This address is added to the effective address in the segmentation unit to produce a 32-bit *linear address*. The paging unit translates the linear address into a 32-bit physical address using a page table.

The segment descriptor and page tables are large and are therefore kept in the main memory. In order to ensure fast address translation, a translation lookaside buffer, as described in Chapter 5, can be used. The segment descriptor tables contain access rights fields as well as segment limit fields to specify the maximum size of a segment. These parameters are managed by the operating system to ensure protection among different application programs that occupy the memory at the same time, giving rise to the name “protected mode.”

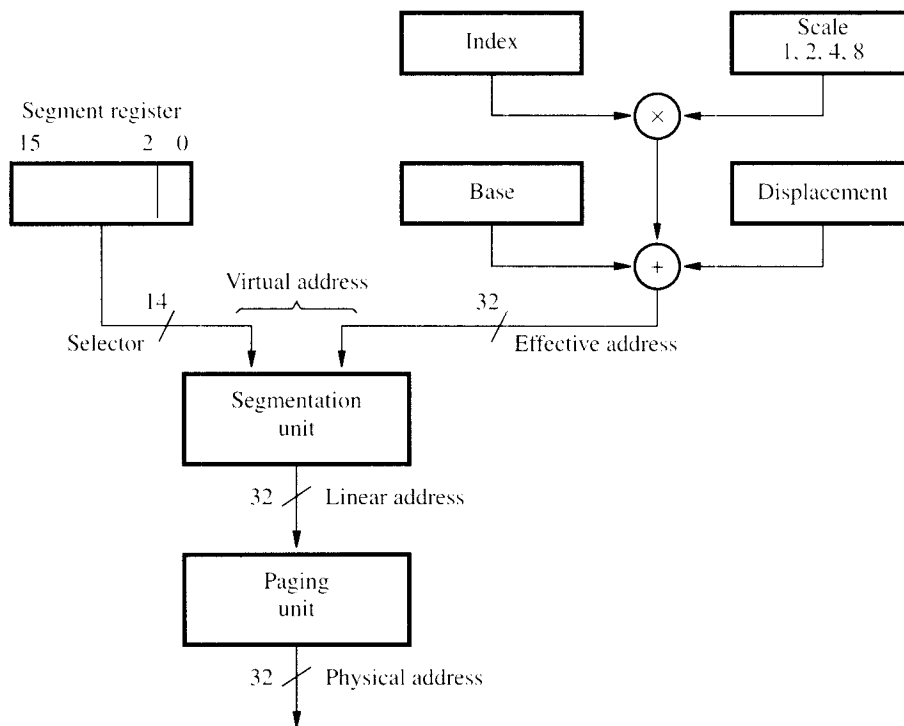


Figure 11.2 Address generation in the IA-32 architecture.

The segmentation and paging features can be used to organize the memory in any of the following ways:

- As a flat, 32-bit address space in which the effective address is used as the physical address
- As one or more variable-length segments (without paging)
- As a 32-bit space divided into one or more 4K-byte pages
- As a structure that combines segmentation and paging

11.3.2 SIXTEEN-BIT MODE

IA-32 processors can operate in a mode in which machine language programs for earlier 16-bit Intel processors (8086 and 80286) can be executed directly. In this mode, only the low-order halves of the processor registers shown in Figure 3.38, labeled AX, CX, are used. The internal address size is 16 bits, and the addressing modes available are a subset of those described in Table 3.3. For example, the value in the index register cannot be scaled in generating effective addresses.

The switch between 32-bit and 16-bit operation can actually be done on an instruction-by-instruction basis. It is also possible to independently choose address and data sizes. For example, full IA-32 addressing mode capability can be used in conjunction with 16-bit data manipulation. When program execution begins, a default mode is set. To switch to the other mode, a prefix byte, not shown in Figure 3.41, is added to an instruction.

11.3.3 80386 AND 80486 PROCESSORS

The 80386 was the first processor to implement the IA-32 architecture described in Chapter 3. It supported both memory segmentation and paging as described in Section 11.3.1.

The 80486 processor was one of the first chips to contain 1 million transistors, roughly the same number as in the Motorola 68040. This processor provides significantly improved performance over the 80386, an advance made possible by expanded circuitry. The 80486 includes both integer and floating-point processing units. The integer unit is fully compatible with the 80386 processor. The floating-point unit implements the IEEE floating-point standard presented in Chapter 6. In 80386-based computers, the same capability was achieved by using a coprocessor chip. Paging and memory management support in the 80486 is the same as in the 80386.

A 4-way, set-associative cache is included in the 80486 for instructions and data. The loading of new information into the cache is enhanced by a burst data transfer mechanism, which enables four 32-bit words to be read as a block and loaded into the cache. The cache has a write-through feature, whereby any data written into the cache are also automatically written into the main memory.

To achieve high performance, the 80486 exploits parallelism and pipelining to a considerable extent. Both the integer unit and the floating-point unit can execute

instructions in parallel. When one instruction is being executed, subsequent instructions are fetched from the memory. Execution of frequently used instructions requires fewer clock cycles than in the 80386; those that load and store data or perform register-to-register operations require only one clock cycle.

11.3.4 PENTIUM PROCESSOR

The performance of the Pentium processor, introduced in 1993 [8], is a significant improvement over that of the 80486. The Pentium has 3 million transistors, compared to 1 million in the 80486, and its computational power is over twice that of the 80486 on integer-intensive benchmark programs and about five times that of the 80486 on floating-point-intensive benchmark programs.

The Pentium processor is a CISC architecture that achieves high performance by using many of the organizational features present in RISC processors, as is done in the 80486 to a lesser extent. In particular, separate on-chip caches of 8K bytes each are used for instructions and data. This superscalar processor, supported by multiple pipelined operation units, can issue two instructions per clock cycle. A 64-bit-wide external data bus allows caches to be rapidly loaded from main memory. The caches are 2-way set associative, with 32-byte blocks. Three independent, pipelined, operational units are included — two for integer operations and one for floating-point operations. Each integer pipeline is five stages deep, and each floating-point unit has eight stages.

The Pentium processor uses a simple form of dynamic branch prediction. It chooses the same direction that it chose the last time the branch was executed. This requires a table of last-branch address values to be kept for each branch instruction. The predicted direction is thus correct for all branches at the end of a program loop after the first branch, until the loop is exited.

11.3.5 PENTIUM PRO PROCESSOR

Introduced in 1995 with a clock rate of 133 MHz, the Pentium Pro [9], [10], provides about twice the performance of a 100-MHz Pentium. Most of the performance gain results from an increased superscalar factor and the capability to execute instructions out-of-order. The superscalar factor, or the maximum number of instructions that can be completed in a clock cycle, is three in the Pentium Pro, compared to two in the Pentium. Pipeline depth in the multiple execution units is twelve, compared to five in the Pentium, and datapath width inside the processor is 64 bits, double that of the Pentium. The Pentium Pro has separate on-chip first level (L1) instruction and data caches of 8K bytes each, the same size as in the Pentium. There is also a second level (L2) cache with a capacity of 256K bytes. It is in the same package as the processor chip, but it is on a separate chip, connected to the processor chip by a 64-bit bus.

Superscalar operation is supported by multiple execution units, including two for integer operations and two for floating-point operations. A major performance-enhancing feature of the Pentium Pro is its ability to execute instructions in an order different from that specified in the program fetched from memory. This feature allows more instructions to be executed in parallel. Of course, adequate control is provided to

ensure that the resulting computations specified by the program are correct. Dynamic branch prediction is implemented in the Pentium Pro, as in the Pentium. This capability allows the processor to look far enough ahead in the instruction stream to take advantage of parallel execution.

External bus monitoring circuits enable the Pentium Pro to be used in multiprocessor systems. These monitoring circuits, and the control actions that they initiate, maintain coherence of common data that may temporarily reside in the caches of different processors. (See Chapter 12 for a discussion of cache coherence.)

11.3.6 PENTIUM II AND III PROCESSORS

The Pentium II processor added the MMX instructions that were briefly described in Section 3.23.2. These instructions facilitate parallel processing of short numbers in multimedia operations on pixels that describe graphical data. The data are operated on in the same eight 64-bit registers used to hold floating-point data. The L1 caches in the Pentium II are double the size of those in the Pentium Pro at 16K bytes each. The off-chip L2 cache holds 512K bytes.

The Pentium III processor introduced the vector (SIMD) instructions discussed in Section 3.23.3. These instructions, called streaming SIMD extension (SSE) instructions [11], are used to efficiently process vector operations on floating-point data. Four 32-bit floating-point operands are packed in each of eight new 128-bit registers, called the XMM registers. The caches are similar to those in the Pentium II, with one major additional option. A version of the Pentium III processor has a 256K-byte L2 cache on the same chip as the processor, providing a higher bandwidth path to the L1 caches.

When they were introduced in 1993, 1995, 1997, and 1999, the Pentium, Pentium Pro, Pentium II, and Pentium III processors had clock rates of 60, 200, 266, and 500 MHz, respectively. Improvements in circuit technology and VLSI fabrication techniques, which have allowed smaller transistor sizes and lower gate delays, have led to clock rates of up to 1 GHz for the current versions of the Pentium III processor.

11.3.7 PENTIUM 4 PROCESSOR

Introduced in 2000, the initial versions of the Pentium 4 have clock rates from 1.3 to 1.5 GHz [7]. The full IA-32 instruction set is supported, including the MMX and SSE instructions. The SSE instructions have been extended (SSE2) to handle two packed 64-bit floating-point numbers or two packed 64-bit integers in the 128-bit XMM registers. The longer integers are useful in implementing the encryption and decryption operations needed in secure data applications. Deeper pipelines — up to 20 stages, compared to 10 in the Pentium III — with shorter stages are one of the factors leading to the higher clock rates, along with improved circuit and fabrication technologies.

There are separate L1 data and instruction caches. The data cache has a capacity of 8K bytes and uses 4-way set associative access to 64-byte cache blocks. The instruction cache is organized to hold decoded instruction execution path segments, called *traces*, which can extend over more than one branch in the original program. If these paths are

repeated, execution is faster. Of course, checks must be made to verify that the same branches are taken when traces are repeated. The term *trace cache* is used to describe this cacheing strategy. The decoded instructions are represented as microoperations. Up to four microoperations are needed to represent an IA-32 instruction. The trace cache can hold a number of execution path segments consisting of a total of 12K microoperations.

The 256K-byte, on-chip, L2 cache is organized into 128-byte blocks and is 8-way set associative. The transfer path between the L2 and L1 cache levels supports a transfer rate of 48 gigabytes/s compared to 16 gigabytes/s in the Pentium III.

The system bus on the Pentium 4 provides a much higher transfer rate than that supported by the Pentium III. The Pentium 4 system bus is 64 bits wide, and transfers can take place at 400 MHz. The transfer rate is therefore 3.2 gigabytes/s as compared to 1 gigabyte/s in the Pentium III.

11.3.8 ADVANCED MICRO DEVICES IA-32 PROCESSORS

Companies other than Intel make processors that implement the IA-32 architecture and compete with the corresponding Intel products for use in personal computers and workstations. Advanced Micro Devices (AMD) [12] has done this for some time. In 2000, AMD's Athlon processor was available with a clock rate of 1.2 GHz, providing performance comparable to the Intel Pentium 4.

The Athlon is a superscalar processor with two levels of cacheing on the processor chip. The L1 cache level has a total capacity of 128K bytes and the L2 cache contains 256K bytes. An interface for double data rate DRAM main memory (see Chapter 5) provides a peak transfer rate of 2.1 gigabytes/s. System I/O bus protocols running at 200 and 266 MHz are provided.

11.4 THE POWERPC FAMILY

In the early 1990s, IBM, Motorola, and Apple collaborated on the development of a RISC-style processor family, the PowerPC [13], [14], for the personal computer and workstation markets. PowerPC processors, produced by both IBM [15] and Motorola [5], have been used in IBM and Apple computers. In general, these processors have architectural features that have provided computing power similar to that of the Intel IA-32 processors over comparable time periods. The first processor implementing the PowerPC architecture, the 601, was produced in 1993. An overview of the instruction set architecture of this family is provided first, followed by a description of some of its processor implementations.

11.4.1 REGISTER SET

There are 32 general-purpose registers and 32 floating-point registers. The floating-point registers are 64 bits long. The IEEE standard is used for representation of floating-point numbers. The PowerPC architecture defines both 32-bit and 64-bit modes of operation.

The size of the general-purpose registers is determined by which of these modes is implemented by a particular processor.

11.4.2 MEMORY ADDRESSING MODES

Memory is byte addressable and is only accessed by Load and Store instructions that transfer data operands between the memory and the processor registers. In keeping with the RISC design style, only simple forms of indexed addressing are used. An effective address is generated by adding the contents of a base register to an index that is either an immediate value contained in the instruction or is the contents of an index register. As an option, the effective address can be loaded back into the base register. This facilitates loading or storing a sequence of operands in contiguous memory locations. Single instructions are available for transferring multiple operands. The Load and Store instructions have a number of versions that provide flexibility in transferring operands of different types and sizes.

11.4.3 INSTRUCTIONS

PowerPC instructions are 32 bits long and have a regular format. The arithmetic and logic instructions use a 3-register format that specifies two operand registers and a destination register for storing the result. A large number of conditional branch instructions are provided. The PowerPC has a MultiplyAdd instruction that performs the operation

$$RD \leftarrow ([RA] \times [RB]) \pm [RC]$$

on floating-point operands in registers RA, RB, and RC. A class of conditional branch instructions decrement a counter and then branch, based on whether or not the decremented value has reached 0. The MultiplyAdd, decrement-and-branch, and multiple operand move instructions, along with optional updating of the base register in indexed addressing, are examples of features that are not normally found in RISC ISAs. These four features are useful in performing multiple arithmetic operations required in signal-processing tasks, efficiently terminating loops, saving and restoring processor registers on procedure entry and exit, and processing lists of data items, respectively. PowerPC designers have incorporated these features without unduly compromising the efficient, streamlined flow of simple, pipelined instructions that is a basic property of RISC machines. We note that all of these features, except for decrement-and-branch, are also present in the ARM ISA in similar forms.

11.4.4 POWERPC PROCESSORS

In the IBM product line, the PowerPC architecture is a successor to the POWER architecture used in the processors of the IBM Risc System (RS)/6000 line of computers. The first implementation of the PowerPC architecture was the 601 processor, which is a transition processor between the two architectures; as such, it implements a superset

of POWER and PowerPC instructions. This allows the 601 to run compiled POWER machine programs as well as PowerPC programs. Later processors in the family are purely PowerPC processors.

PowerPC 601 Processor

The 601 processor chip, containing 2.8 million transistors, was first used in IBM desktop machines. The 601 is a 32-bit processor, intended for notebook, desktop, and low-end multiprocessor systems. Different versions were available with processor clock rates of 50, 66, 80, and 100 MHz.

The PowerPC 601 has a 32K-byte cache on the processor chip for holding both instructions and data. The cache is organized in 8-way associative sets. Three independent execution units are provided: an integer unit, a floating-point unit, and a branch-processing unit. Up to three instructions can be issued for execution in a clock cycle for superscalar operation. The 601 has four pipeline stages for integer instructions and six for floating-point instructions.

PowerPC 603 Processor

The 603 processor also has a 32-bit processing width. Intended for notebook and desktop computers, it is a low-cost, low-power processor, consuming about 3 watts of electrical power at 80 MHz. The five execution units provided can operate in parallel, so the instruction issuing and control hardware, which can issue up to three instructions per clock cycle, is somewhat more complex than in the 601. The on-chip cache is divided into two 8K-byte sections for separate, temporary storage of instructions and data.

PowerPC 604 Processor

The 32-bit 604 processor was designed for higher performance than was available in either the 601 or the 603; both integer and floating-point speeds are approximately double those in the 601 and 603 processors. The 604 achieves this performance level with a 100-MHz clock rate and a superscalar capability for issuing up to four instructions per clock cycle. The processor has six independent execution units: three integer units, a floating-point unit, a memory load/store unit, and a branch-processing unit. Personal computers and midrange workstations were the intended market for this processor.

PowerPC 620 Processor

The 620 processor implements the full 64-bit PowerPC architecture and supports superscalar performance. It was targeted for high-end desktop computers, servers, transaction processing systems, and multiprocessor systems.

Like the 604, the 620 has six independent execution units, and up to four instructions can be completed in a clock cycle. The actual rate of processing instructions in a particular program is enhanced by the processor's ability to execute instructions out of order. Dynamic branch prediction is used, and the processor chip contains both instruction and data caches. Each cache holds 32K bytes and is organized in 8-way associative sets.

MPC7450 Processor

The 601, 603, 604, and 620 were the first implementations of the PowerPC by IBM. After this 6XX line, Motorola implemented the 7XX and 7XXX lines of PowerPC processors with their MPC prefix on the labeling. The latest processor in the MPC7XXX line is the MPC7450, introduced in early 2001 with clock rates up to 733 MHz. It is used in the Apple Power Mac G4 computer line.

The MPC7450 is a superscalar processor with a 7-stage pipeline. Up to four instructions per clock cycle can be issued into the function units. There are eleven such units: A load/store unit, a branch unit, four integer units, a floating-point unit, and four units that perform parallel arithmetic operations on packed vector data operands. Motorola uses the name AltiVec [16] for these latter units.

The AltiVec hardware performs parallel operations on vector data operands similar to the way that Intel Pentium processors perform MMX and SSE operations, as described in Sections 3.23.2 and 11.3.6. The packed data operated on by AltiVec instructions are located in thirty-two 128-bit vector registers, separate from the general-purpose and floating-point registers. A vector register can hold sixteen 8-bit integers, eight 16-bit integers, four 32-bit integers, or four single-precision (32-bit) floating-point numbers. Vector load and store instructions are used to transfer data between memory and the vector registers. AltiVec instructions speed up multimedia and signal-processing operations. One of the instructions is a Multiply-Accumulate instruction that multiplies corresponding elements in two vector registers and adds the products to corresponding elements of a third vector register. This operation is common in digital signal processing operations. Vector dot product instructions are also included.

The on-chip L1 cache level comprises separate 32K-byte instruction and data caches. These caches are 8-way set associative. An L2 cache is also included on the processor chip. It contains 256K bytes and is also 8-way set associative. Transfers between the L1 and L2 caches are done over a 256-bit path at the processor clock rate. An off-chip L3 cache is accessed over a 64-bit bus. It can be configured for 1M- or 2M-byte capacities.

11.5 THE SUN MICROSYSTEMS SPARC FAMILY

The SPARC architecture was developed by Sun Microsystems Corporation to be a scalable architecture. It is the basis for a series of processors that provide increasingly higher performance as implementation technology and organizational innovations develop. The basic instruction set architecture has remained the same. SPARC processors are intended for the high-performance workstation and server market. They are designed so that they can be used in multiple processor systems where the processors share a common main memory. Such systems are described in Chapter 12.

The SPARC architecture is a RISC-style architecture with a 3-register, 32-bit, fixed length instruction format. Two source operand registers and a destination register are specified in an instruction. All instructions that perform operations on data manipulate the operands in processor registers. There are 32 general-purpose registers for integers and addresses, and 32 registers for floating-point operands. The only instructions that

access memory are the load and store instructions that transfer operands between the registers and the memory.

The first implementations of the SPARC architecture, in 1987, handled 32-bit addresses and 32-bit data. The latest version of the architecture, version 9, handles both addresses and data as 64-bit values, and is implemented by the UltraSPARC series of processors. Instructions continue to be 32 bits wide, and the programmer's model of all registers remains the same. Backward compatibility has been maintained, that is, the UltraSPARC processors can execute machine code from earlier versions of the architecture.

The SPARC architecture was introduced in Chapter 8. The UltraSPARC II processor was used there to illustrate how pipelining is implemented in a high-performance processor. The UltraSPARC family, which includes UltraSPARC I, II, and III, features multiple execution units and superscalar performance. In addition to the basic SPARC instruction set, a number of special instructions have been introduced to support graphics and multimedia applications. They are called the *visual instruction set* (VIS). VIS instructions provide parallel vector operations on graphics pixels or digital signal samples packed into 64-bit words. The VIS instructions are similar to the Intel MMX and SSE instructions (Part III of Chapter 3 and Section 11.3.6) and the Motorola AltiVec instructions (Section 11.4.4). Successive members of the UltraSPARC family offer increasingly higher performance, with higher clock speeds, faster memory and I/O interfaces, and larger and more sophisticated cache organizations. For example, the UltraSPARC I was manufactured using 0.5-micron CMOS technology and used a clock frequency of 167 MHz [17]. Its successor, the UltraSPARC II, has a very similar 9-stage pipeline organization, but achieves higher performance because of the faster 0.25-micron technology used [18]. It operates with clock frequencies in the range 250 to 480 MHz.

The most recent member of the family, the UltraSPARC III, uses a 14-stage pipeline [19]. There are four integer execution units and three floating-point units, including processing for the VIS instructions. The UltraSPARC III is manufactured in 0.18-micron technology. Clock frequencies are in the range 750 to 900 MHz, and future models are intended to run as high as 1.5 GHz. The on-chip L1 data cache contains 64K bytes and the instruction cache contains 32K bytes. They are both 4-way set associative and operate with 32-byte blocks. The external L2 cache is direct mapped and can be configured for 4M- or 8M-byte capacity. The UltraSPARC III provides extensive support for use in multiprocessor configurations that can potentially have hundreds of processors.

The microSPARC Family

Another family of processors based on the SPARC architecture is called microSPARC. Members of this family are 32-bit processors based on version 8 of the SPARC architecture specifications. They are intended for low-cost uniprocessor applications. Some of these processors, such as microSPARCIIep, include a PCI interface and memory controller on the processor chip, and are well suited for embedded applications. The fact that these microprocessors are fully compatible with the UltraSPARC processors offers considerable advantage to developers of embedded applications. The

software intended for a given application can be developed and tested on powerful workstations and then downloaded to the target processor in the final stages of development.

11.6 THE COMPAQ ALPHA FAMILY

Digital Equipment Corporation introduced the Alpha architecture in 1992 as the successor to the 32-bit VAX family [20]. Digital Equipment was acquired by the Compaq company in 1998. Compaq [21] produces a line of high-end workstations and server systems that use Alpha processors, labeled with the numbering sequence 21X64, with $X = 0, 1,$ and 2 .

The Alpha architecture is a RISC design with 64-bit address and data sizes. There are 32 general-purpose and 32 floating-point registers. Multiple pipelined operation units are used in all 21X64 processors to achieve superscalar instruction execution rates, enhanced by both static and dynamic branch prediction. Separate on-chip data and instruction caches are used.

A basic goal of pipelined processor design is to keep logic depth short in each pipeline stage to minimize the stage delay for any given implementation technology, permitting a high clock rate. An important design characteristic of the Alpha architecture is that it uses only simple instruction formats and addressing modes to achieve short pipeline stage delays. Only 32-bit and 64-bit aligned loads and stores are permitted between the L1 cache and the processor, minimizing the delay in those transfers.

11.6.1 INSTRUCTION AND ADDRESSING MODE FORMATS

The Alpha ISA has only four instruction types, all 32 bits long:

Operate — Integer, floating-point, and byte-manipulation operations are included in this class. These instructions use a three-operand format, with operands contained in processor registers or in an immediate field of the instruction.

Memory — Load/store operations use register plus displacement indexed addressing as the only addressing mode.

Branch — Conditional branch instructions contain a displacement value that specifies the direction and distance of the branch target address relative to the program counter. There is no condition code register; condition codes are optionally written into a general-purpose register by operate instructions. This register is then named by branch instructions that need to test the codes. Unconditional branch instructions use the named register to hold the updated value of the program counter as the return address if the branch is a subroutine call.

Call-PAL — Privileged Architecture Library (PAL) instructions perform operating system functions not available in user mode. These privileged instructions can access hardware resources, that is, processor state registers, that are not accessible by the normal instruction set. PAL routines also contain instructions that do not exist in the defined Alpha instruction set. They service interrupts and manipulate memory management unit registers.

11.6.2 ALPHA 21064 PROCESSOR

The first implementation of the Alpha architecture, the 21064, is a 200-MHz chip with 1.7 million transistors that dissipates 30 watts of power [22]. It contains 8K-byte L1 instruction and data caches. Both caches are direct mapped and have 32-byte blocks. An external L2 cache can be configured with a capacity between 128K and 8M bytes. The memory management unit has separate translation lookaside buffers for instruction (12 entries) and data (32 entries) accesses.

A maximum of two instructions can be issued per clock cycle. Four independent processing units are used: an integer unit, a floating-point unit, a branching unit, and a memory load/store unit. The pipeline depths in these four units are seven, ten, six, and seven, respectively. The first four stages are common and can handle two instruction streams in parallel.

11.6.3 ALPHA 21164 PROCESSOR

The 21164 processor was introduced in 1994 [23]. It provides roughly double the performance of the first 21064 processors. The transistor count in the 21164 is 9.3 million, and 50 watts of power are dissipated at a clock rate of 300 MHz. An on-chip, unified, 96K-byte L2 cache is provided, in addition to the 8K-byte L1 instruction and data caches. The L2 cache is 3-way set associative and has 64-byte blocks. An off-chip L3 cache can be configured with a capacity between 1M and 64M bytes.

The maximum instruction issue rate is four instructions per clock cycle, double that of the 21064. There is one more functional unit than in the 21064. It is used to manage the L2 and L3 caches. Pipeline depths are similar. Memory management hardware includes a 48-entry translation lookaside buffer for accessing instructions, and the buffer for accessing data has 64 entries.

11.6.4 ALPHA 21264 PROCESSOR

The 21264 is the latest processor in the 21X64 line [24]. It was introduced in 1998 with a 500-MHz clock rate. In early 2001, versions running at up to 850 MHz were available. There are 15 million transistors in the processor chip.

The cache arrangement is significantly different from earlier Alpha processors. The L1 instruction and data caches are much larger at 64K bytes each. They are both 2-way set associative caches. The unified L2 cache is off-chip, and it can be configured with a capacity between 1M and 16M bytes. There is no L3 cache level. The increased hit rates in the larger L1 caches lead to an overall decrease in memory access latencies, even though the L2 cache is off-chip.

Another significant difference between the 21264 and earlier Alpha processors is the ability to issue instructions to the functional units in an out-of-order sequence, as discussed in Chapter 8. This increases the sustainable superscalar instruction execution rates achievable on typical programs. The maximum instruction issue rate is four per cycle, the same as in the 21164. But functional unit resources are increased by replicating

sections of both the integer and floating-point units and adding a new functional unit for processing video data. This increase in instruction execution hardware, coupled with the higher clock rates and the out-of-order instruction issue capability, leads to a performance level about twice that of the 21164.

11.7 THE INTEL IA-64 FAMILY

Since the mid-1990s, Intel and Hewlett-Packard have been jointly developing a 64-bit microprocessor architecture called IA-64 [25]. The first processor that implements this architecture is called the Itanium (earlier code-named the Merced). The IA-64 architecture is completely different from the IA-32 architecture, which has been Intel's 32-bit architecture from the 80386 onward, through the continuing Pentium line of processors. Intel is expected to continue to produce processors for both the IA-32 and IA-64 architectures. Programming aspects of the IA-64 architecture are described in references [26] and [27].

The IA-64 architecture has a 64-bit address space and 64-bit integer and floating-point formats. The 3-register RISC-like instruction format occupies 41 bits. There are three 7-bit register fields for addressing the 128 general registers or the 128 floating-point registers. A 6-bit field specifies conditional execution of the instruction as described below. The remaining 14 bits of the instruction specify the OP code.

11.7.1 INSTRUCTION BUNDLES

A distinctive feature of the IA-64 architecture is that three 41-bit instructions are grouped into a 128-bit *bundle*, along with a 5-bit field called the *template* which specifies compiler-derived information about how instructions can be executed in parallel. For example, one of the template codes indicates the location of a *stop*, which marks the end of a group of instructions that can be executed in parallel. Such a group may extend over a number of bundles. Information in the templates is used by the processor to schedule the parallel execution of instructions on multiple functional units to achieve superscalar operation. This feature of the IA-64 architecture is called *Explicitly Parallel Instruction Computing* (EPIC). EPIC can be considered as an extension of the concept of *Very Long Instruction Word* (VLIW) instruction set design [28]. In VLIW architectures, each instruction specifies a number of possibly different operations that can be applied in parallel to independent data operands.

11.7.2 CONDITIONAL EXECUTION

A major aspect of the IA-64 architecture is the use of conditional execution of instructions, called *predication*. A 6-bit *predicate* field in each instruction selects one of 64 one-bit *predicate flags* contained in the processor. These flags effectively replace the condition code flags in conventional processors. If the named flag is equal to 1, the

instruction is executed; otherwise, it is not. Actually, the instruction is processed through the instruction pipeline, but its results are written into the destination location only if the predicate flag is 1. This feature is similar to the conditional execution of instructions in the ARM architecture, described in Part I of Chapter 3.

Conditional execution of instructions increases the rate of executing program instructions by removing conditional branches in certain situations. For example, a short, forward, conditional branch can be eliminated by conditionally executing the code block between what would otherwise be the location of a conditional branch instruction and its target location. The predicate flag guarding execution of each instruction in the code block is set by a test or compare instruction ahead of the block.

A similar opportunity for performance enhancement occurs in generating IA-64 code for an *if-then-else* construct. Figure 11.3a shows conventional machine code for executing the Add instruction if the contents of registers R1 and R2 are equal, or executing the Subtract instruction if they are not. The corresponding IA-64 code is shown in Figure 11.3b. The IA-64 CompareEqual instruction operates as follows. Predicate flag P1 is set to 1 if the contents of R1 and R2 are equal; otherwise, it is set to 0. Flag P2 is set to the complement of P1. The Add instruction is executed if $P1 = 1$, and the Subtract instruction is executed if $P2 = 1$. Double semicolons indicate the positions of stops. In this example, the Add and Subtract instructions between the two double semicolons can be scheduled for parallel execution. Only one of them will have its result actually written into the destination register specified, as determined by the values of P1 and P2. The instruction execution pipelines will not be stalled if the values of P1 and P2 are determined before the write stages of the Add and Subtract instructions are reached. In

```

                Compare      R1,R2
                Branch≠0    ELSE
THEN:  Add      R3,R4,R5
                Branch      NEXT
ELSE:  Subtract R6,R7,R8
NEXT:  ...

```

(a) Conventional code

```

CompareEqual  P2,P1 = R1,R2 ::
(P1) Add      R3,R4,R5
(P2) Subtract R6,R7,R8 ::
NEXT:  ...

```

(b) IA-64 code

Figure 11.3 Implementing *if-then-else* code in the IA-64 architecture.

addition to these types of performance enhancing features, the IA-64 also uses branch prediction and speculative execution as discussed in Chapter 8.

11.7.3 SPECULATIVE LOADS

In order to mitigate against the delays introduced by register load instructions that may need to access main memory, a special form of load, called a *speculative load*, can be generated by the compiler. This load is placed ahead of where it would normally appear in a conventionally compiled program. This increases the chance that it will be in the register when it is needed, avoiding any memory access delay. A check must be made that it actually is there when it is about to be used. Special care must be taken in handling speculative loads that are moved ahead of predicted branches.

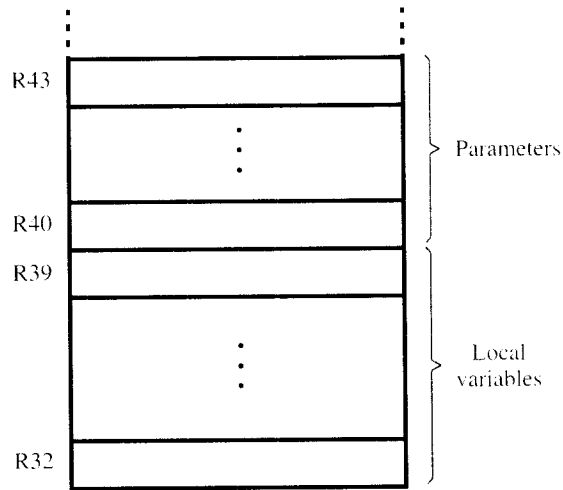
11.7.4 REGISTERS AND THE REGISTER STACK

The IA-64 architecture specifies 128 general registers that can be used to hold 64-bit integers or 64-bit addresses. There are also 128 registers for holding double-precision (64-bit) floating-point numbers. Two single-precision numbers can be packed into one register. In addition, there are eight 64-bit registers for holding subroutine call/return linkage addresses.

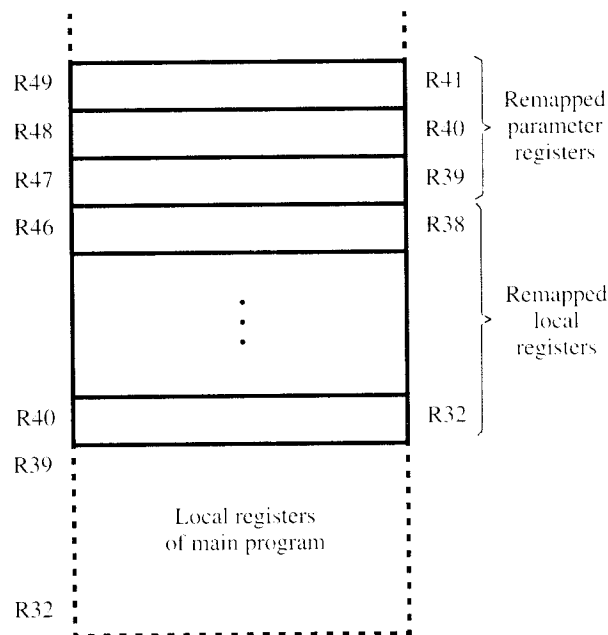
The first 32 of the 128 general registers, R0 through R31, are used as normal data or address registers. The remaining 96 registers, R32 through R127, are handled as a *register stack* for holding the local variables of subroutines and the parameters passed between calling and called routines. This register stack effectively replaces the processor stack, implemented in memory, as described in Section 2.9.1. The register stack is managed in such a way that registers need not be saved/restored to/from the memory as a sequence of nested subroutines is called, as described below. This assumes that the total local and parameter variable register space required by all of the subroutines does not exceed 96. If that happens, the processor control hardware automatically “spills” a portion of the register stack into the memory to create the needed excess register space, and it automatically loads that portion back into the register stack as returns are executed.

Register renaming is also managed automatically by the processor so that all routines — the main routine and called subroutines — always refer to their local registers from R32 upward, even though the actual physical registers may be different. (Another version of register renaming was discussed in Chapter 8.) An overlap region of the high end of a caller’s local registers with the low end of the called routine’s local registers is used to implement parameter passing.

Figure 11.4 shows an example of how the register stack is managed when the main program calls a subroutine. The register ordering in this figure shows a stack that grows upward. This is done for easy comparison with stack illustrations in Chapter 2, where stacks grow toward lower memory addresses. Registers R0 through R31 are available to all routines and can be considered as holding global variables. They are not shown in the figure. The main program is assumed to use the eight registers R32 through R39 for its local variables, and the four registers R40 through R43 to pass parameters to a



(a) Active register area in the main program after execution of `Alloc 8,4`



(b) Active register area in the subroutine after execution of `Alloc 7,3`

Figure 11.4 Register stack allocations for local variables and parameter passing in IA-64.

subroutine. This use of registers is declared to the processor hardware by executing the instruction

```
Alloc 8,4
```

in the main program. Figure 11.4a shows this situation. After the main program calls the subroutine, the subroutine executes the instruction

```
Alloc 7,3
```

to declare that it requires seven local registers, including the four used to receive parameters passed from the calling routine, as well as three registers to pass parameters to a second subroutine. The ten (physical) registers R40 through R49 are remapped by the processor hardware so that they can be referenced as (logical) registers R32 through R41 by the subroutine. The part of the register stack that is active during execution of the subroutine is shown in Figure 11.4b. The information needed to do this dynamic remapping at execution time is derived from the values in the Alloc instruction executed by the main program. When the subroutine performs the return to the main program, the active portion of the register stack returns to the state shown in Figure 11.4a. The register stack implements a version of multiple *register windows*, as employed in the Berkeley RISC designs [29], [30], and in the UltraSPARC II processor (see Chapter 8).

Another form of register renaming, called *register rotation*, is also used in the IA-64. It is used to overlap the execution of successive iterations of a loop, assuming that there are no iteration-to-iteration data dependency restrictions. Normally, successive iterations of a loop use the same registers, those named in the loop body. In the IA-64 architecture, the compiler generates code for a single copy of the loop body in a form that allows the hardware to automatically rename the registers so that different physical registers are used on successive iterations of the loop. This permits the instruction scheduling and issuing hardware to start successive iterations before previous iterations have completed. This technique for reducing total loop execution time is called *software pipelining*. It differs from loop unrolling, in which replicated copies of the loop body are placed in the machine instruction code.

11.7.5 ITANIUM PROCESSOR

The Itanium processor is the first implementation of the IA-64 architecture [31]. It has a relatively large number of replicated functional units for the different types of operations: integer, floating-point, and multimedia (like the MMX operations of the IA-32 architecture, described in Part III of Chapter 3). Superscalar execution is achieved by the ability to issue up to six instructions (two 3-instruction bundles) on each 800-MHz clock cycle into the 10-stage pipeline. The functional units are: 4 integer units, 4 floating-point units, 4 multimedia (MMX) units, 2 load/store units, and 3 branch units. The integer register bank has 8 read ports and 6 write ports in order to exchange data simultaneously with the multiple function units.

There are three levels of cache units. Both L1 and L2 caches are on the same chip as the processor, and L3 is implemented on separate chips in the same cartridge package with the processor. The L1 cache level consists of separate instruction and data caches,

each containing 16K bytes. These caches are 4-way set associative, and they have 32-byte blocks. The instruction cache can deliver two 3-instruction bundles (256 bits) to the processor per clock cycle. The L2 cache contains 96K bytes. It is 6-way set associative and has 64-byte blocks. The L3 cache contains a total of 4M bytes, with 64-byte blocks. It is 4-way set associative and communicates with the L2 cache over a 128-bit internal bus connection at the processor clock rate, providing a transfer rate of 12.8 gigabytes/s.

Interactions between the caches and between the L1 cache and the processor are organized in a way that minimizes the effect of processor pipeline stalls and cache misses on the average instruction execution rate. Some aspects of these interactions are worth noting. There is a decoupling buffer between the L1 instruction cache and the processor. It can contain up to eight 3-instruction bundles. This allows prefetching of instructions from the L1 cache into the buffer to continue when the processor stalls in issuing instructions. Conversely, the processor can continue to fetch and issue instructions from the buffer when a cache miss occurs in the prefetching process. There is also a buffer between the L1 and L2 caches to allow the prefetching of instructions from L2 into L1. It is twice the size of the buffer between the L1 cache and the processor instruction issue hardware. The L1 cache only feeds the integer register bank. Floating-point operands are loaded into the floating-point registers directly from the L2 cache.

There is a 64-bit system bus connecting the package containing the processor and the caches to other system components such as main memory and I/O devices. It can support a 266-MHz transfer rate, which is 2.1 gigabytes/s.

The external bus controller can accommodate the direct connection of up to four Itanium processors in a multiprocessor configuration. The controller handles the cache coherence operations required when the multiple processors share common external memory units, as described in Chapter 12.

11.8 A STACK PROCESSOR

All processors discussed in this book use general-purpose registers to hold data operands. Instructions can access them in any desired order. Some years ago, the Hewlett-Packard Company designed and manufactured a computer called the HP3000, whose main architectural feature is an instruction set that is keyed to processing operands held in a stack data structure. Access to operands in a stack is restricted to only those operands residing at the top of the stack, and results are always returned to the top of the stack. This type of organization is not appropriate for current RISC and CISC processor designs that are highly parallel. In these processors, simultaneous access to several operands in a large register set is required for high performance. Nevertheless, the HP3000, and the earlier series of B5500, B6500, and B6700 computers produced by the Burroughs Corporation, which also featured stack-oriented processing, are historically important as commercial implementations of stack computing. The way these machines process arithmetic expressions is both interesting and elegant. We illustrate the main ideas by describing the HP3000 instruction set and addressing modes. Our discussion concentrates on only the features that characterize the stack organization of this computer.

11.8.1 STACK STRUCTURE

The HP3000 is a 16-bit computer. Its memory contains program instructions and data in separate domains: instructions and data cannot be intermixed except for immediate data that can be used in programs. Hardware registers are used as pointers to the program and data segments, as shown in Figure 11.5.

Three registers specify the program segment. The program base (PB) and the program limit (PL) registers indicate the memory area occupied by the program, and the program counter (PC) points to the current instruction. Each of these registers contains the appropriate 16-bit address.

The data segment is divided into two parts — the stack and the data area. Five 16-bit pointers are used to delineate and access these memory locations. The contents of the data base (DB) register denote the starting location of the stack. The stack grows in the higher-address direction. If the top element of the stack is at location i , then the next element pushed onto the stack will be at location $i + 1$. This contrasts with other stacks discussed in the book, where it has been assumed that they grow in the direction of decreasing addresses. The address of the top element in the stack, also called the

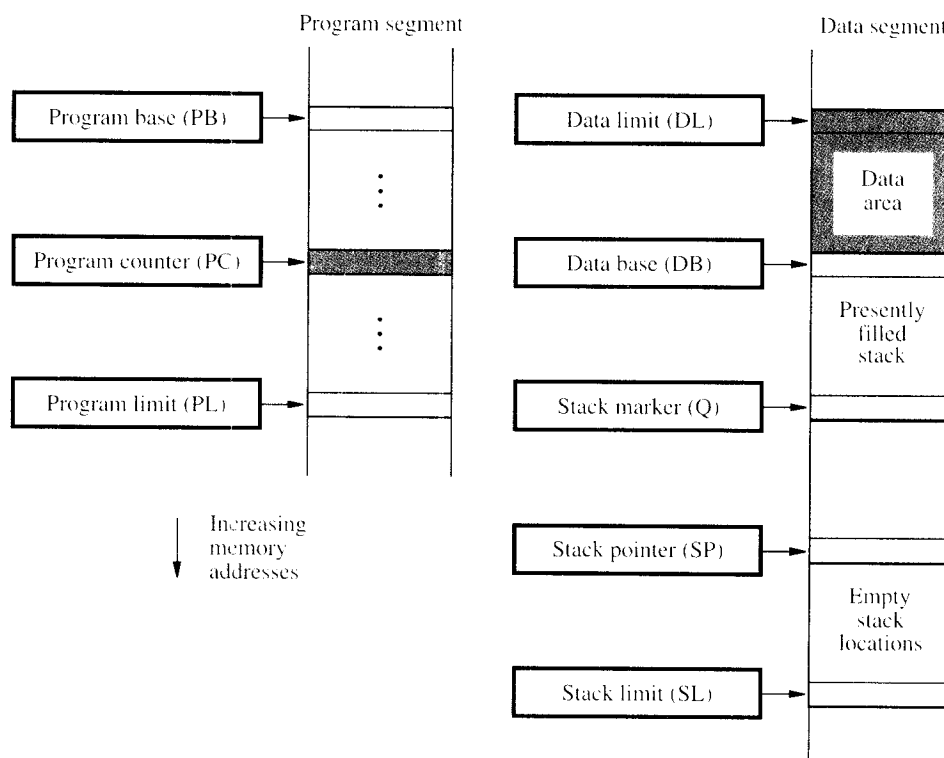


Figure 11.5 Program and data segment organization in the HP3000.

top of stack (TOS), is stored in the 16-bit stack pointer (SP). The SP is not actually a single hardware register, as we explain shortly, but it can be thought of that way. It is incremented or decremented when data elements are pushed onto or popped off the stack. From the user's point of view, it functions as any other 16-bit pointer register. The upper limit of the stack is defined by the contents of the stack limit (SL) register. Therefore, the stack is allowed to grow until $[SP] = [SL]$. Any attempt to extend the stack past the limits defined by DB and SL is prevented by the hardware. The data area extends from the location immediately preceding the location pointed to by the DB register to the limit specified in the data limit (DL) register.

The pointer registers specify the stack's current size, maximum size, and location in memory. The stack is thus a dynamic structure that can be easily changed. Figure 11.5 shows one other pointer, the stack marker (Q) register. This register denotes the starting point for the data stack of the current procedure. Actually, Q points to the fourth word of a four-word entry in the stack, called the *stack marker*, that facilitates passing control between procedures. The Q register serves a role similar to the frame pointer register described in Chapter 2. When a procedure must be suspended, for example, as a result of an interrupt, the information needed to allow proper return to the suspended procedure is placed onto the stack in the form of a stack marker.

The first word of the stack marker stores the current contents of an index register, and the second word contains the return address. The return address information is actually stored as the difference between the value in the PC, which points to the next instruction to be executed in the current procedure, and the contents of the PB register. By storing the difference, instead of the absolute value, programs can be moved out of the memory and later returned to a different place in the memory. The new area in memory is pointed to by loading a new value in the PB register. The third word saves the status information contained in the status register, and the fourth word stores the distance between this stack marker and the one immediately preceding it.

Figure 11.6 shows one stack marker, denoted k , that was placed on the stack at the time Procedure $_k$ was initiated, and another stack marker, denoted $k + 1$, that is placed onto the stack when a new procedure, Procedure $_{k+1}$, is initiated. When the new procedure is completed, the machine transfers control to the previous procedure using the data in the stack marker $k + 1$. At that time, the Q register must be set to point to the fourth word of stack marker k . This is readily accomplished because the distance between the stack markers is stored as a part of each marker. Also, the SP is set to point to the location immediately preceding stack marker $k + 1$. As a result, SP points to the top of the stack used by Procedure $_k$, thus restoring the situation that existed at the time Procedure $_{k+1}$ was invoked. This technique can be used to nest any number of procedures. Parameter passing between procedures also uses the stack.

In addition to the pointer registers, HP3000 computers have other hardware registers used in the internal organization of the machine. The only two of these that are visible to the programmer are the index and status registers, which function in essentially the same ways as similarly named registers in most other computers. Note, however, that there are no general-purpose registers available to the programmer. Instead, data are manipulated using the stack as temporary storage, as we show in an example in the next section.

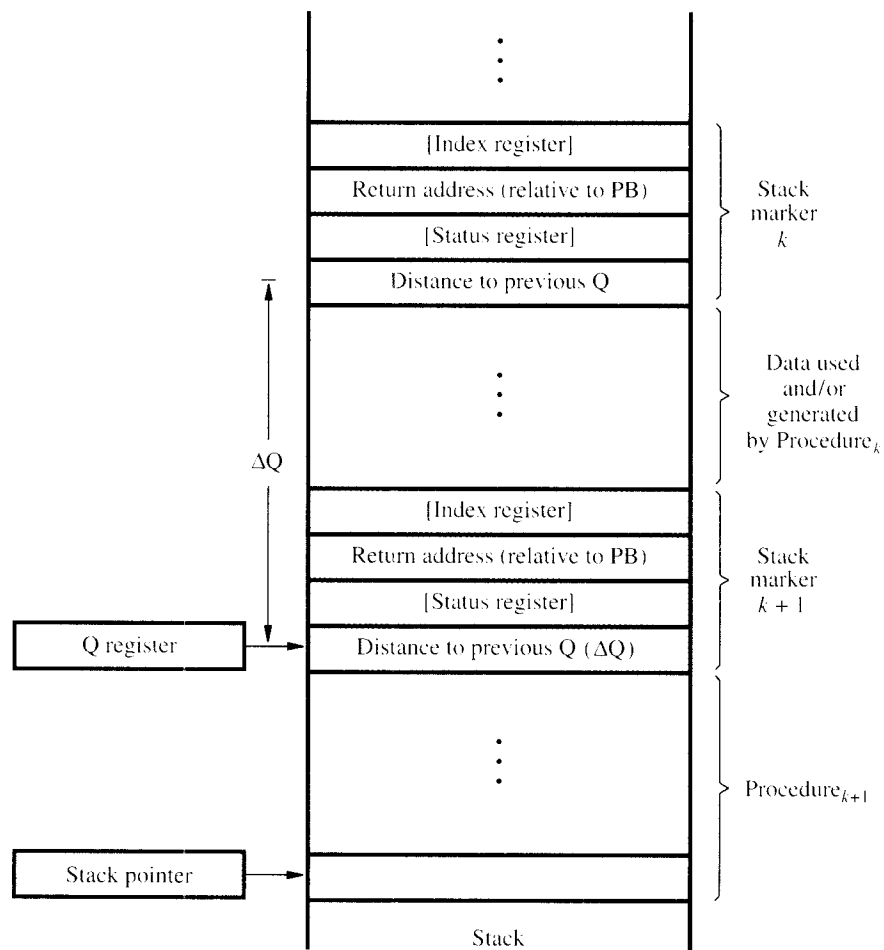
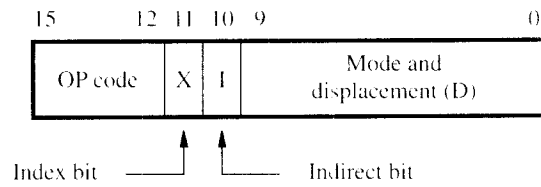


Figure 11.6 Stack markers in HP3000.

11.8.2 STACK INSTRUCTIONS

As a basic strategy, stack computers perform operations on data that occupy the top few locations of the stack. Furthermore, the results generated are left on the stack. This assumes that there are instructions that can move data between the stack and the memory.

The HP3000 has a variety of instructions that are all 16 bits long. Most of the instructions involve the stack in some way, and typically either the operands, operand addresses, or other relevant parameters reside in the stack. This allows great flexibility in using the 16-bit code space of the instructions. There are 13 major classes of instructions. Instead of describing the full HP3000 instruction set, we restrict our attention to the classes that illustrate the stack organization of the machine. Let us first consider the Memory Address instructions, whose format is shown in Figure 11.7. Eleven valuations



Mode	Bit pattern										Effective memory address
	b_9	b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	
PC + relative	0	0	← D →								$[PC] + D$
PC - relative	0	1	← D →								$[PC] - D$
DB + relative	1	0	← D →								$[DB] + D$
Q + relative	1	1	0	← D →							$[Q] - D$
Q - relative	1	1	1	0	← D →						$[Q] - D$
SP - relative	1	1	1	1	← D →						$[SP] - D$

Figure 11.7 Memory Address instruction format in HP3000.

of the 5-bit OP-code field are used to specify instructions in this class. The Memory Address instructions include:

- LOAD** Push a specific memory word onto the stack.
- STOR** Pop the top word of the stack (TOS) into a specified memory location.
- ADDM** Add a specified memory word to TOS, and replace the TOS operand with the resultant sum.
- MPYM** Multiply a specified memory word with TOS, and replace the TOS operand with the least significant word of the product.
- INCM** Increment a specified memory word.

These instructions specify the memory operand in the relative address mode, in which addresses are given relative to the contents of the PC, DB, Q, or SP registers. The 10-bit mode and displacement field indicates the mode and the magnitude of the displacement, as the figure shows. The displacement is not the same in all modes because the displacement field varies from 6 to 8 bits. Index and indirect bits specify whether indexed or indirect addressing or both are to be performed. These are the only addressing modes that can be used to address operands in the data area of Figure 11.5.

The second class of instructions is the Move instructions, which reference either one or two memory operands. These instructions can move words or bytes from one memory location to another, compare two strings of bytes in the memory, or scan a byte string until a particular byte value is found. Memory addresses are again computed in the relative mode. The displacement is not given explicitly within the instruction.

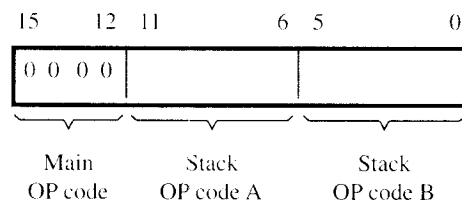


Figure 11.8 Format for Stack instructions in HP3000.

however, but is included as data in the stack. Moreover, addresses can only be specified relative to the program or data bases, that is, the contents of the PB or DB registers. A good example of this class of instructions is the basic MOVE instruction. It transfers k words from one memory location to another, where

- The first stack element, TOS, specifies k .
- The contents of the second stack element give the address of the first source memory location relative to either PB or DB.
- The contents of the third stack element give the address of the first destination memory location relative to DB.

This instruction can be represented within a 16-bit code space because most of the addressing data and the length parameter are given in implicitly specified stack locations. These data must be loaded onto the stack before the instruction can be executed.

Next, we consider the Stack instructions, whose format is shown in Figure 11.8. This class of instructions is identified by four zeros in the high-order bit positions. The remaining 12 bits are available to specify particular instructions and are split into two 6-bit fields, each of which can be used to specify a distinct operation. The 6 bits allow as many as 64 distinct stack operations to be defined. This number is large enough to accommodate a variety of stack operations. An instruction specifying one stack operation uses 10 bits (the main OP code plus stack OP code A) and disregards the remaining 6 bits. When the remaining bits specify a second stack operation (using stack OP code B), that operation is performed after the first operation is completed. In this way, two stack operations can be packed within a single instruction. Such efficient utilization of the instruction code space is possible only because addressing data and operands are not included explicitly as part of an instruction.

Some examples of Stack instructions are

ADD Add the contents of the top two words on the stack, delete them from the stack, and push the sum onto the stack.

CMP Compare the contents of the top two words on the stack, set the condition codes accordingly, and delete both words from the stack.

DIV Divide the integer in the second word of the stack by the integer in TOS. Replace the second word with the quotient and the word in TOS with the remainder.

DEL Delete the top word of the stack.

Many instructions of this type are provided, although some are more complicated. A Divide Long (DIVL) instruction, for example, divides a double-word integer in the second and third elements of the stack by the integer in the first element. Then these three words are deleted, and the remainder and quotient are pushed onto the stack to become the first and second elements, respectively. We use the term “instruction” somewhat loosely in this discussion of stack instructions. It would be more accurate to refer to Add and Divide operations, for example, since these two operations can be specified within a single instruction. However, it is more customary to speak in terms of instructions when describing such actions, and it is appropriate to describe the preceding technique as packing two instructions into one. Such packing is possible only when two consecutive stack operations are to be performed. In other cases, OP code B is left unused.

So far, we have emphasized only one advantage of compressing instructions, that of the low code-space requirements. Another advantage stems from the reduced number of memory accesses because two instructions are effectively fetched as part of one 16-bit word. We must remember that, during execution of a stack instruction, operands in the stack must be accessed, and this requires memory accesses if the stack resides in the memory.

To illustrate the role of the stack as temporary storage for intermediate results in arithmetic processing, we consider a simple example. Figure 11.9 shows how the arithmetic expression

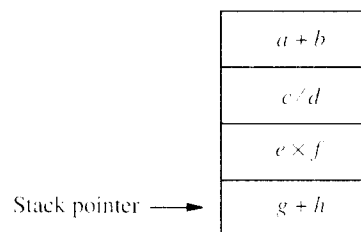
$$w = \frac{(a + b)}{c/d + (e \times f)/(g + h)}$$

is evaluated. We assume that the values of the variables a, b, \dots, h are not available at the top of the stack. They are stored in memory locations with addresses A, B, \dots , H, and can be accessed with the addressing mechanism given in Figure 11.7. Furthermore, assume all operands are integers whose sizes are such that only single-length products need to be considered. The figure shows 13 processing steps that must be performed. The required operations follow the order obtained by scanning the numerator and denominator of the expression from left to right. In our notation, the top element of the stack (TOS) is denoted as S. Thus, the operation $S \leftarrow [S] + [B]$ means that the contents of TOS and operand B are added, and the sum replaces the value in TOS. The operation $S \leftarrow [S - 1]/[S]$ indicates that the contents of the second element in the stack are divided by the contents of TOS. The two operands are deleted from the stack and the quotient and remainder are pushed onto the stack.

The HP3000 machine instructions needed to perform the necessary computation are shown in the figure. Their function is described earlier in this section. Most steps can be implemented with a single instruction, except for the division operation. The DIV instruction replaces the dividend and the divisor with the quotient and the remainder, respectively. Because we are only interested in the quotient, we use the DEL instruction to delete the remainder from the stack. Whenever two consecutive Stack instructions are encountered, they can be combined into one 16-bit instruction, as we explain earlier. All intermediate results are stored on the stack. Figure 11.9*b* shows the top elements of the stack after step 9 is completed.

Step	Operation performed	Machine instruction	
1	$S \leftarrow [A]$	LOAD A	
2	$S \leftarrow [S] + [B]$	ADDM B	
3	$S \leftarrow [C]$	LOAD C	
4	$S \leftarrow [D]$	LOAD D	
5	$S \leftarrow [S - 1] / [S]$	DIV DEL	} combined
6	$S \leftarrow [E]$	LOAD E	
7	$S \leftarrow [S] \times [F]$	MPYM F	
8	$S \leftarrow [G]$	LOAD G	
9	$S \leftarrow [S] + [H]$	ADDM H	
10	$S \leftarrow [S - 1] / [S]$	DIV DEL	} combined
11	$S \leftarrow [S - 1] + [S]$	ADD	
12	$S \leftarrow [S - 1] / [S]$	DIV DEL	} combined
13	$W \leftarrow [S]$	STOR W	

(a) Operations to be performed and the necessary machine instructions



(b) Temporary results stored in the stack after step 9

Figure 11.9 Stack usage in processing the expression
 $w = (a + b) / [c / d + (ef) / (g + h)]$.

11.8.3 HARDWARE REGISTERS IN THE STACK

Accessing memory locations is one of the most critical time constraints in a computer. The time needed to read an operand from the memory is longer than the time required to perform operations in processor registers. This is the main reason for including

general-purpose registers and caches in the processor. In the case of stack computers, the temporary storage function of the general-purpose registers is provided through the stack mechanism. If the stack is implemented entirely in the memory, the processor must make frequent memory accesses because all temporary storage locations are part of the stack.

The possibility of implementing the entire stack with hardware registers could be expensive and somewhat inflexible. A compromise between an all-register or an all-memory implementation of the stack is possible, however, if most of the stack is located in the memory and its top few elements are held in hardware registers in the processor. The time to access the stack is then reduced because most accesses involve only the top few elements and therefore only require register transfers within the processor. In the HP3000 computer, four registers contain the top four elements of the stack.

Including hardware registers in the stack implies that the true top of the stack (TOS) is often one of the registers. This means that the SP does not necessarily point to a memory location. To keep track of where the top elements of the stack are at any given time, the SP function is implemented by two registers. A 16-bit stack in memory (SM) register contains the address of the highest memory location presently occupied by the stack, and a 3-bit register (SR) indicates whether zero, one, two, three, or four top elements of the stack are presently contained in the hardware registers. Thus, the value [SP] is

$$[SP] = [SM] + [SR]$$

This value is equal to the address in the memory where the top element of the stack would be if all elements of the stack were in the memory. This structure is illustrated in Figure 11.10. The programmer does not have to be aware of the inclusion of hardware registers in the stack. For the programmer's purpose, only one pointer exists — the stack pointer, SP.

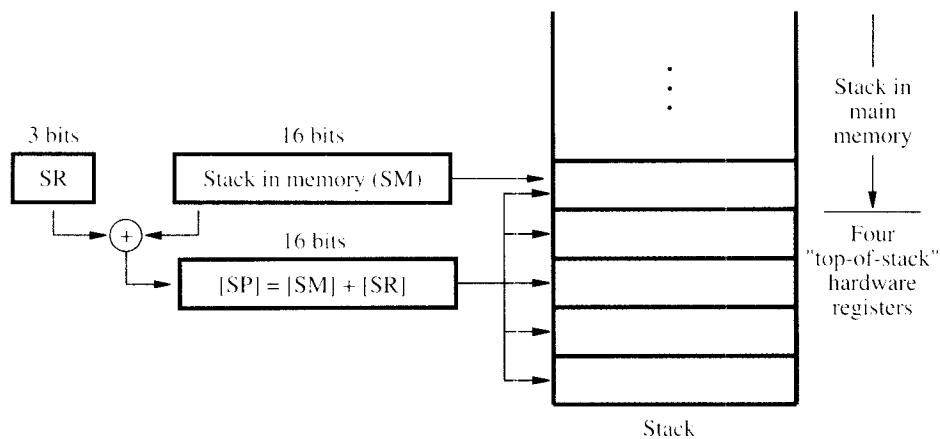


Figure 11.10 Top-of-stack structure in the HP3000.

11.9 CONCLUDING REMARKS

High-performance processors all employ on-chip caches for instructions, data, and address translation operations. They have multiple, independent, pipelined processing units and they have the capability to issue more than one instruction per processor clock cycle to these units, possibly in out-of-order sequence. Sophisticated methods for dynamic branch prediction and speculative execution of alternate program paths, made possible by predicated execution of instructions, increases overall performance. These design features are fundamental to achieving high execution rates for programs, independent of whether the instruction sets and addressing modes follow the RISC or CISC design approaches. Optimizing compilers that translate high-level language programs into efficient machine language code are also critical to achieving high performance. Many factors must be considered when evaluating the performance of a computer, and standardized benchmark programs are commonly used for this purpose, as discussed in Chapter 1.

PROBLEMS

- 11.1 How is conditional execution of ARM instructions (see Part I of Chapter 3) related to predicated execution of IA-64 instructions? Comment on similarities and differences.
- 11.2 The 16-bit Thumb instruction subset of ARM instructions is intended for compact program encoding. Estimate the number of Thumb instructions needed to program the evaluation of the arithmetic expression shown in Figure 11.9. Assume that the Thumb subset has an appropriate divide instruction (which it does not) for the purposes of this question. How does the estimated Thumb instruction count compare to the HP3000 program instruction count in Figure 11.9a?
- 11.3 Discuss the similarities and differences between the Motorola 680X0 family and the Intel 80X86 family of processors, up to the 68040 and the 80486 versions.
- 11.4 The 68030 microprocessor has a 256-byte instruction cache and a 256-byte data cache. Is this better than having a 512-byte instruction cache and no data cache? What are the advantages and disadvantages of these two alternatives? Answer the same two questions for a unified 512-byte instruction and data cache.
- 11.5 Intel IA-32 processors have special instructions for dedicated I/O operations, as described in Part III of Chapter 3. Motorola 680X0 processors use only memory-mapped I/O. What are the advantages and disadvantages of memory-mapped I/O compared to dedicated I/O?
- 11.6 Discuss the relative merits of addressing modes in the Motorola 680X0 and Intel 80X86 processors. In particular, discuss how the addressing modes in each processor facilitate program relocation, implementation of a stack, accessing an operand list, and manipulating character strings.

- 11.7** Section 11.3.1 explains that an IA-32 processor can view the memory as being organized in four different ways, depending on how segmentation and paging are used. Give some examples of situations in which each of the four possibilities is beneficial.
- 11.8** Write an ARM, 68000, or IA-32 program to evaluate the arithmetic expression in Figure 11.9. How does your program compare to the one in the figure with respect to the number of machine instructions required? Assume that the standard ARM instruction set has an appropriate divide instruction (which it does not) for the purposes of this question.
- 11.9** In Alpha processors, only 32-bit and 64-bit aligned loads and stores are directly handled in the datapath between the cache and the processor. Sketch the combinational logic network that would be required in a 32-bit-wide datapath to permit loading any one of the four bytes of a 32-bit quantity into the low-order byte position at the destination side of the path.
- 11.10** Compare the handling of the register stack in the IA-64 processors to that of the processor (memory) stack in Chapter 2. In particular, what are the counterparts of the Chapter 2 stack pointer, SP, and frame pointer, FP, in the IA-64 scheme?
- 11.11** Give a general description of the hardware needed to support execution of the Alloc X.Y instruction used for managing the IA-64 register stack. Assume that small registers and adders are available. How are they used?
- 11.12** The Alpha 21264 processor has a much different arrangement of caches than the 21164. Why is the arrangement in the 21264 better? That is, under what circumstances do programs execute more quickly on the 21264, based only on the effects of caching? Include more than an observation on hit rates in your answer.
- 11.13** Show how the expression

$$w = a \left[(b \times c) + (d \times e) + \frac{f \times g}{h \times i} \right]$$

can be evaluated in an HP3000 computer.

- 11.14** In an HP3000 computer, Procedure_{*i*} generates eight words of data, DI₁, . . . , DI₈, which are stored in the stack. After these words are placed in the stack, but before the completion of Procedure_{*i*}, a new procedure, Procedure_{*j*}, is called. It generates 10 words of data, DJ₁, . . . , DJ₁₀, which are also stored in the stack. Then another procedure, Procedure_{*k*}, is called, which places three words of data in the stack. Show the contents of the top words of the stack at this time.
- 11.15** Show how the expression

$$w = (a + b)(c + d) + (d \times e)$$

can best be evaluated by the HP3000, ARM, Motorola 68000, and IA-32 computers. The values of variables *w*, *a*, *b*, *c*, *d*, and *e* are stored in memory locations. The following assumptions are made. The addresses do not reference successive locations. Direct memory addressing in the DB+ relative mode is used in the HP3000. Absolute/Direct

memory addressing is used in the 68000 and IA-32 computers, and Relative addressing is used in the ARM computer. All products are single length.

- 11.16** What is the largest number of stack locations occupied during execution of the program in Figure 11.9?
- 11.17** Repeat Problem 11.16 for the HP3000 programs in Problems 11.13 and 11.15.

REFERENCES

1. S. Furber, *ARM System-on-Chip Architecture*, 2nd ed., Addison-Wesley, Great Britain, 2000.
2. *IEEE Micro*, vol. 17, no. 4, eight articles on ARM, July/August 1997.
3. ARM web site: arm.com
4. D. Tabak, *Advanced Microprocessors*, McGraw-Hill, New York, 1991.
5. Motorola web site: motorola.com
6. J. Circello, et al., "The Superscalar Architecture of the MC68060," *IEEE Micro*, vol. 15, no. 2, April 1995, pp. 10–21.
7. Intel web site: intel.com
8. D. Alpert and D. Avnon, "Architecture of the Pentium Microprocessor," *IEEE Micro*, vol. 13, no. 3, June 1993, pp. 11–21.
9. R.P. Colwell and R.L. Steck, "A 0.6-Micron BiCMOS Processor with Dynamic Execution," *Proceedings of the International Solid State Circuits Conference*, February 1995.
10. "A Tour of the P6 (Pentium Pro) Microarchitecture," Intel Corporation, 1995.
11. S.K. Raman, V. Pentkovski, and J. Keshava, "Implementing Streaming SIMD Extensions on the Pentium III Processor," *IEEE Micro*, vol. 20, no. 4, July/August 2000, pp. 47–57.
12. Advanced Micro Devices web site: amd.com
13. *Communications of the ACM*, vol. 37, no. 6, eight articles on the PowerPC, June 1994.
14. *IEEE Micro*, vol. 14, no. 5, five articles on the PowerPC, October 1994.
15. IBM web site: ibm.com
16. K. Diefendorff, et al., "Altivec Extensions to PowerPC Accelerates Media Processing," *IEEE Micro*, vol. 20, no. 2, March/April 2000, pp. 85–95.
17. M. Tremblay and J.M. O'Connor, "UltraSparc I: A Four-Issue Processor Supporting Multimedia," *IEEE Micro*, vol. 16, no. 2, April 1996, pp. 42–50.
18. Sun Microsystems web site: sun.com
19. T. Horel and G. Lauterbach, "UltraSPARC III: Designing Third Generation 64-bit Performance," *IEEE Micro*, vol. 19, no. 3, May/June 1999, pp. 73–85.

20. *Digital Technical Journal*, vol. 4, no. 4, issue on Alpha, 1992.
21. Compaq web site: compaq.com
22. E. McLellan, "The Alpha AXP Architecture and 21064 Processor," *IEEE Micro*, vol. 13, no. 3, June 1993, pp. 36–47.
23. J.H. Edmondson, et al., "Superscalar Execution in the 21164 Alpha Microprocessor," *IEEE Micro*, vol. 15, no. 2, April 1995, pp. 33–43.
24. R.E. Kessler, "The Alpha 21264 Microprocessor," *IEEE Micro*, vol. 19, no. 2, March/April 1999, pp. 24–36.
25. *IEEE Micro*, vol. 20, no. 5, six articles on the IA-64 architecture and the Itanium processor, September/October 2000.
26. C. Dulong, "The IA-64 Architecture at Work," *COMPUTER*, vol. 31, no. 7, July 1998, pp. 24–32.
27. R. Krishnaiyer, et al., "An Advanced Optimizer for the IA-64 Architecture," *IEEE Micro*, vol. 20, no. 6, November/December 2000, pp. 60–68.
28. R.P. Colwell, et al., "A VLIW Architecture for a Trace Scheduling Compiler," *IEEE Transactions on Computers*, vol. 37, no. 8, August 1988, pp. 967–979.
29. D. Patterson and D. Ditzel, "The Case for the Reduced Instruction Set Computer," *ACM SIGARCH Computer Architecture News*, vol. 8, no. 6, October 1980, pp. 25–33.
30. M. Katevenis. *Reduced Instruction Set Computer Architectures for VLSI*, MIT Press, Cambridge, MA, 1985.
31. W.A. Samaras, N. Cherukuri, and S. Venkataraman, "The IA-64 Itanium Processor Cartridge," *IEEE Micro*, vol. 21, no. 1, January/February 2001, pp. 82–89.

