issues of data caches concern organization and the method of supporting write operations (discussed in Section 2.2.6).

## 2.4.1 *Published Data Cache Miss Data*

Data-addressing patterns have poorer temporal and spatial locality than instruction-addressing patterns. Unlike instruction and unified caches, until recently, there has been little published $P_{miss}$ data for data caches. Table 2.15 shows the $P_{miss}$ data for data caches excerpted from [GEE91, GEE93] for SPEC benchmarks. The combined floating point and integer (called overall) $P_{miss}$ values are shown for an organization ($S$, $SE$, 1, 16).

The information in Table 2.15 shows a significant increase in $P_{miss}$ for direct caches as compared to other organizations of the same size. For all other organizations, the value of $P_{miss}$ is only sensitive to cache size. Note the tradeoff derived from doubling the size of a data cache. For a small cache ($S = 64$, $SE = 1$) it is better to double the number of sectors rather than the number of sets, if possible. However, for a large cache, there is no significant difference in $P_{miss}$ for the two options if the original cache design is not direct ($SE = 1$). These observations are similar to those made about instruction caches.

| | Number of Sets | | | | | | |
|---|---|---|---|---|---|---|---|
| No. of Sec./Set | 64 | 128 | 256 | 512 | 1K | 2K | 4K |
| 1 | .255 | .216 | .170 | .114 | .087 | .069 | .057 |
| 2 | .188 | .155 | .104 | .074 | .059 | .048 | .039 |
| 4 | .134 | .098 | .071 | .058 | .047 | .038 | .028 |
| 8 | .091 | .069 | .057 | .047 | .038 | .028 | .017 |

Block size is 16 bytes.

**TABLE 2.15** Data cache $P_{miss}$, SPEC benchmarks, overall.

## 2.4.2 *Contemporary Data Caches*

Data caches, used in split cache topologies, are becoming common with contemporary processors. In this section, a representative sample of data cache designs are described. In some cases the data cache organization is the same as the instruction cache and, in other cases, different.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000110

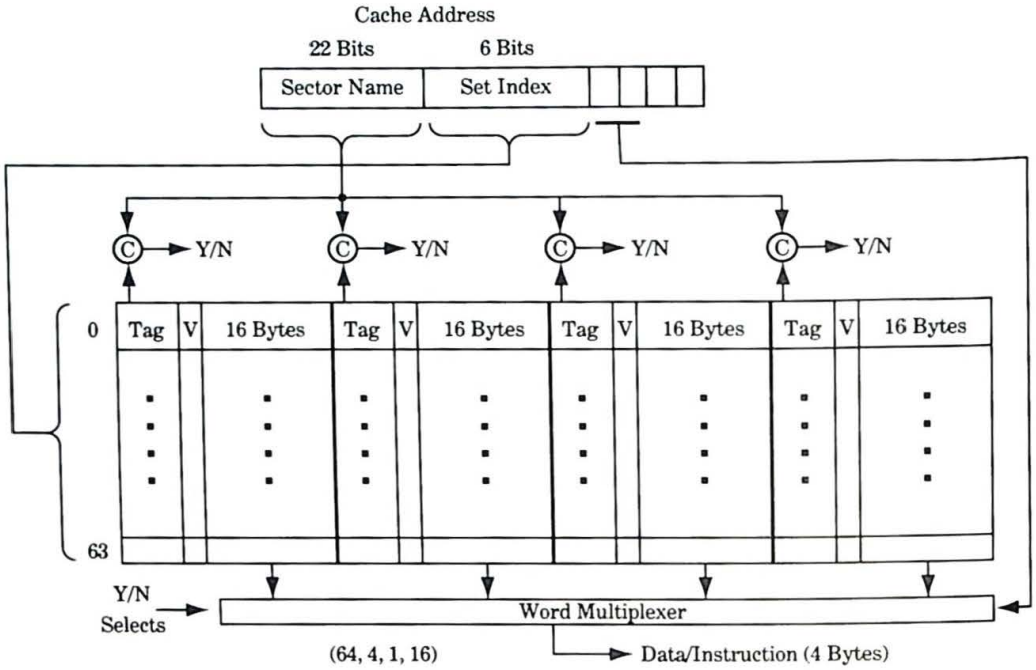



**FIGURE 2.18** Motorola 68040 data and instruction cache.

## MC68040 Data Cache

The designers of the MC68040 chose to have a symmetrical cache [MOTO89]; that is, the instruction and data caches are identical. Each cache is 4 Kbytes, organized as (64, 4, 1, 16) [MOTO89] and shown in Figure 2.18. This cache has 22 tag bits per sector, various control bits, and a 16-byte block. Two bits of the address select one of four words in the sector-block that is passed to the processor.

We speculate that the reason behind the design decision to use the same organization for the data and instruction caches is that the design effort is less costly due to reduced the design cycle time and design cost of the processor. The $P_{\text{miss}}$ data of Table 2.14 and Table 2.15 for SPEC benchmarks indicate that, for two 4-Kbyte caches, identical organizations provide reasonable $P_{\text{miss}}$ for both caches. Tweaking the organizations would be of little benefit.

The split caches of the MC68040 communicate to the memory system over a common bus that serves cache misses. A second-level cache is not anticipated because of the relatively large size of the on-chip caches and the resulting low $P_{\text{miss}}$.

For caches with one block per sector (sector-block), a replacement policy is needed to select a sector-block for eviction for a data cache, or overwriting for an instruction cache, when all of the tag comparisons

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000111

indicate occupied sector-blocks. Such a replacement policy is illustrated with the MC68040 data cache. The replacement policy follows two steps.

1. Locate the first invalid sector-block and replace it.

2. If step #1 does not succeed, a pseudorandom algorithm selects one of the valid sectors (blocks) and replaces that sector. There is a 2-bit count that is incremented once for each half-sector access; this counter provides the pseudorandom pointer to the sector to be replaced.

## Motorola MC88100 Data Cache

The Motorola MC88100 architecture specifies two paths to the main memory—one path for instructions and another for data. In support of this memory organization, the MC88200 provides two caches of equal size and organization. Note that this is a chip separate from the processor chip and that instruction and data references pass from the processor to this chip. The two caches can be addressed with either virtual or real addresses. The block diagram is shown in Figure 2.19 [MOTO90a].

This cache organization is (256, 4, 1, 16) with the usual tag bits for

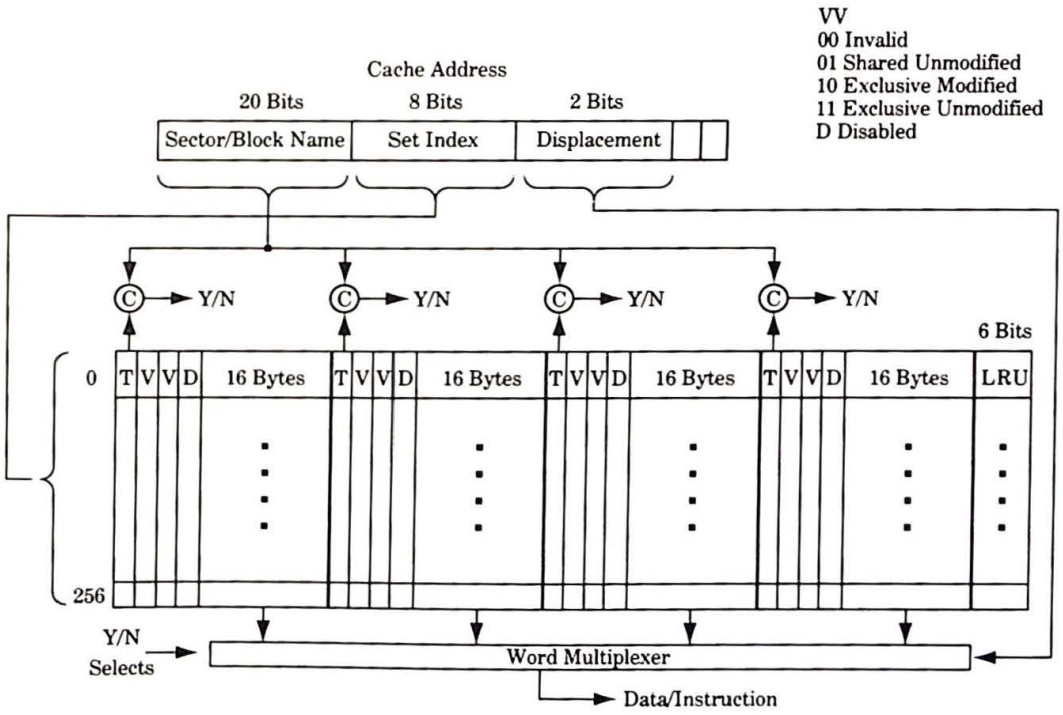



**FIGURE 2.19** Motorola 88200 cache.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000112

each sector-block. In addition, there are two V bits and a D bit that have the coding shown in the figure. These bits are used to manage the cache and preserve coherency in a multiprocessor configuration (discussed further in Chapter 4). In addition, each set has six bits that maintain an LRU status of the four sectors within each set. These bits support an LRU replacement policy for the cache. Upon a miss, if one or more of the valid bits are not set, that sector-block is used for the new sector. However, if all of the valid bits are set, the LRU mechanism selects the sector-block to be written back if in the write back mode or to be overwritten if it is in the write through mode.

As the cache is filled, the six pseudorandom bits, numbered L0 through L5, are set as follows.

Sector-block1 is *more recently used than* Sector-block0, Set L0.
Sector-block2 is *more recently used than* Sector-block0, Set L1.
Sector-block2 is *more recently used than* Sector-block1, Set L2.
Sector-block3 is *more recently used than* Sector-block0, Set L3.
Sector-block3 is *more recently used than* Sector-block1, Set L4.
Sector-block3 is *more recently used than* Sector-block2, Set L5.

Identification of the LRU sector-block can be made by examination of these six bits. It can be shown, for example, that for the LRU bits 111001 (L5–L0) sector-block3 is the most recently used and is the candidate for replacement.

### IBM RS/6000 Data Cache

The IBM RS/6000 data cache organization, shown in Figure 2.20, is (128, 4, 1, 128) with 64 Kbytes [BAKO90]. The management policy is late select, allocate, write back (store in). The 128 sets are divided into four groups of 32 sets; the five LSBs of the set index access the four groups in parallel. The two MSBs of the set index are provided after they are translated via the TLB (discussed in Chapter 4 and Fig. 4.5) and provide a late select of one of the four groups of sets.

The write back strategy is supported by 128-byte *store back buffers* (SBB). When a block is to be written to the main memory, all 128 bytes of the sector-block are transferred in parallel to the SBB and then sent to the main memory in eight, 16-byte chunks.

When a reload of a cache sector-block is required, the strategy is *fetch bypassed (load through)* (see Section 2.2.4) with the sector-block being loaded into the 128-byte *cache reload buffers* (CRB) and the miss causing word passing to the processor. The sector-block resides in the CRBs until an idle cache cycle permits the sector-block to be transferred in parallel into the cache. The duty cycle of a data cache is less than that of an instruction cache; as a result, idle cycles will occur. If a read

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000113

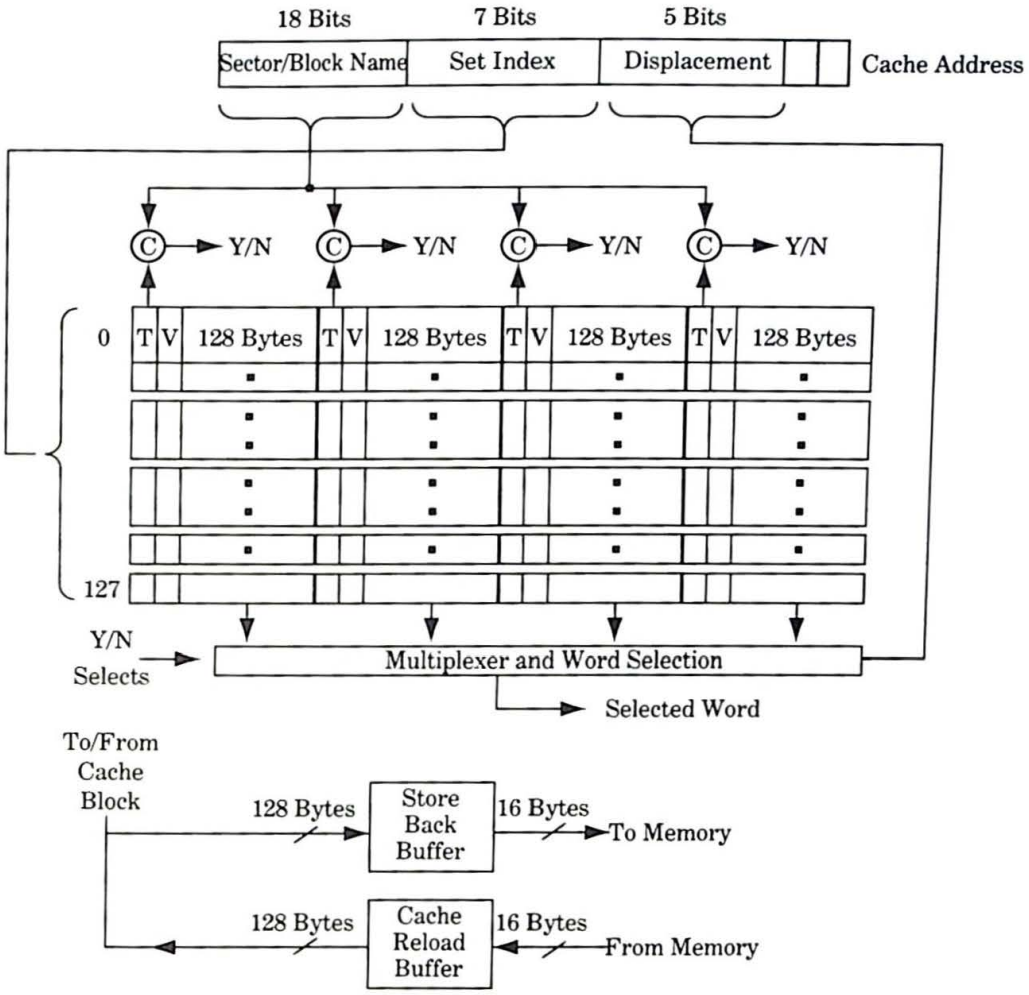



**FIGURE 2.20** IBM RS/6000 data cache.

reference is made to this sector-block before it is transferred into the cache, the fetch is satisfied from the CRB. This action precludes the cache being locked out from other references during the time the new block is waiting for an idle cache cycle. Nonblocking loads are possible because the CRB is disjoint from the cache.

There is only one CRB; therefore, if a second cache miss occurs before the CRB has been transferred, the second miss is not served until the first load is complete. The CRB acts like a one sector-block cache with tags that permits reads from it. It appears from [MATI89] that a write miss uses an allocate strategy on the CRB. The sector-block is fetched into the CRB, modified, moved into the SBB, and then written back to the main memory.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000114

### 2.4.3 *Data Queues*

Section 2.3.2 describes the use of prefetch queues to provide a constant instruction fetch rate for variable-length instructions and to hide memory latency and cache miss transport time. Similar techniques have been used for data fetches as well. The same two problems are present with data fetches, namely, variable-length data types and the desire to hide memory latency and cache miss transport time.

The first instance, known to me, of the use of a data queue is with the IBM S/360/91 [AND67]. When an instruction that requires a memory reference is decoded, the effective address is computed, the read request is sent to the interleaved memory, and the fetched data is sent to one of two of the six-deep data queue.

There are separate queues for integer and floating point data. The predecoded integer and floating point instructions have also been placed on queues and intercept the queued data in the execution units. The data queues are deep enough to hide most, if not all, memory latencies.

Renewed interest in data queues is found in [EICK93], a paper that reports on recent research on the use of data queues. A major problem with data queues is that an effective address is required to access data to be placed in the queue. With pipelines, there can be true dependencies on the effective address, leading to a problem. As with branches, a prediction of the effective address must be used that may prove to be incorrect. For example, one prediction strategy is to use the same effective address used the last time the instruction was executed. However, this predicted effective address may be incorrect, and the prefetch is not effective and unnecessarily loads the bus with traffic. The [EICK93] paper describes effective address prediction strategies and assesses their effectiveness.

## 2.5 Unified Caches

A unified cache holds both instructions and data and is a manifestation of the von Neumann architecture. It provides a single port to the main memory, via the cache, for both instruction and data references. The organization of a unified cache is selected to best match the address sequences of combined instruction and data reads and must support data write operations.

Early research and design of caches focused on unified caches as seen in the work of [BLOO62, LIPT68, LEE69]. During this early period, processors were either hard-wired or microprogrammed with little overlap in execution. Thus, instruction and data reads did not usually contend for access to the cache. Without instruction and data contention, the single access port toe unified cache is satisfactory. This is not the case

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000115

with pipelined processors. An early processor that could have contention is the UNIVAC LARC, an experimental pipelined processor that could simultaneously fetch both instructions and data [ECKE59]. However, the use of a large general-purpose register file reduced the number of contentions.

Bloom et al. [BLOO62] described the first cache when studying design alternatives for a high-performance computer. The cache organization is (1, 16, 1, 1). Each sector-block of the cache contained the tag, valid bits, and one computer word of data. This cache is named a *look aside buffer*, a term later associated with virtual memory. Replacement of a sector-block is on an LRU basis and determined by the charge on a capacitor associated with each sector-block. Other early work on caches was performed by Project MAC, sponsored by ARPA (a.k.a. DARPA).

The IBM S/360/85 is the first known commercial implementation of a cache in a large-scale computer [LIPT68]. Subsequent to the implementation of the IBM S/360/85, unified caches have become pervasive and are employed in many computers. Only with realizable, practical VLSI technology have unified caches given way to split caches, which are discussed further in Section 2.6. In many contemporary designs a unified cache is the second level to a first-level split cache or used with systems having a first-level instruction buffer or queue.

The IBM S/360/85 cache is used as an example of an early select cache and is shown in Figure 2.21 [MATI77] and organized as (1, 16, 16, 64). The sector name in the cache address field is associatively searched against the tag bit fields. If the sector is resident in the cache, the 4-bit sector base address is gated out and concatenated with the block index and displacement fields to form a 14-bit data memory address.

Each of the 1024-byte sectors is divided into sixteen, 64-byte blocks. Each of the 256 blocks has a valid bit that is collected into a validity bit register, which is a special register that is addressed by the block index concatenated with the translated set base address. Because the data memory is implemented with core, integration of the valid bits would have been difficult. A cache miss fetches one block of 64 bytes, and the write policy is write through on hit and no-write-allocation miss.

Another early unified cache that used the early select method is the IBM S/360/195. This cache is organized as (128, 4, 1, 64) [MATI77] and is shown in Figure 2.22.

The set index selects one of 128 sets having four sectors containing the sector base address for accessing the data memory. Note that the seven set index bits are displaced to the left and there are nine tag bits. The reason this displacement is carried out is to make each of the 128 sets match the 8-Kbyte page size of the virtual memory. However, this cache organization is not a pure congruent system because of this arrangement of address bits. This cache is early select and has a

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000116

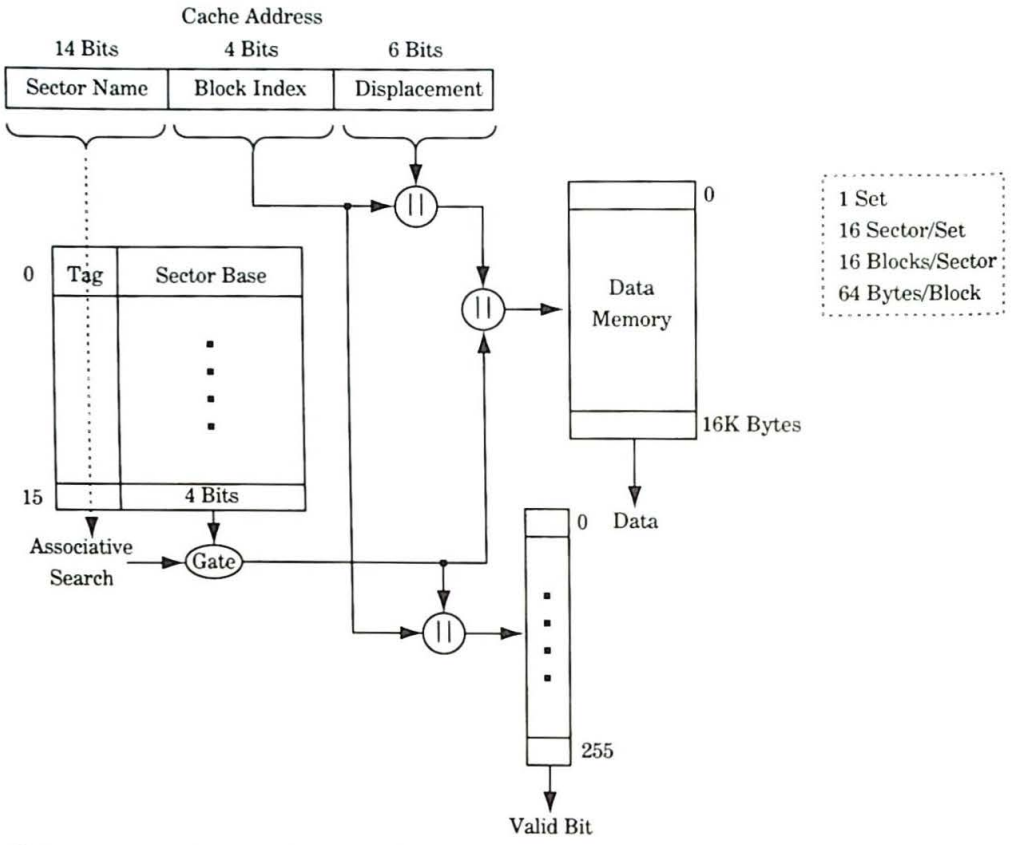



**FIGURE 2.21** IBM S/360/85 unified cache.

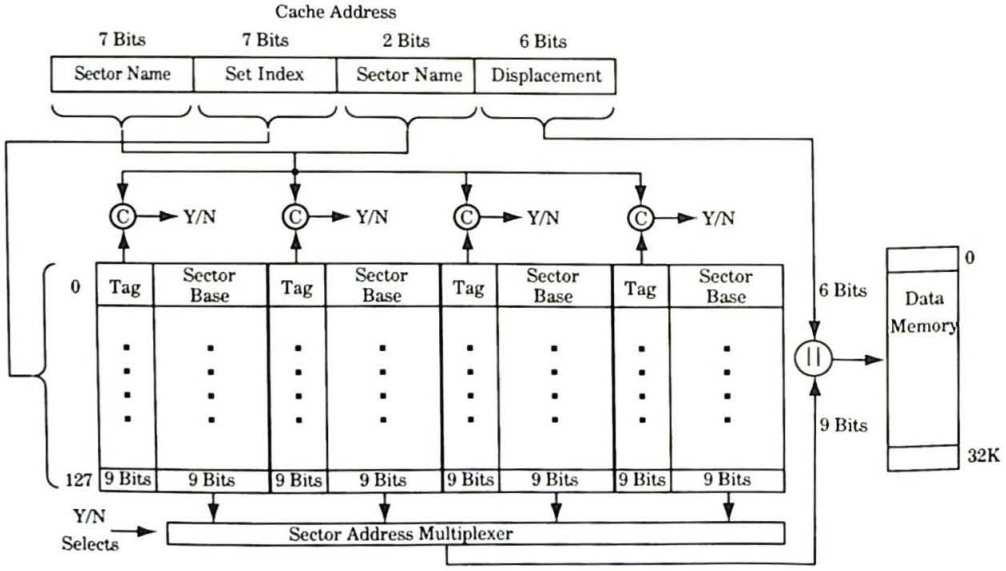



**FIGURE 2.22** IBM S/360/195 unified cache.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000117

three-level write buffer providing nonblocking write operations. The write buffer is associatively searched to resolve true dependencies on memory (discussed in Chapter 8).

The PowerPC 601 uses an on-chip unified cache. This cache supplies data to the register file and instructions to the instruction queue. The cache is organized as (64, 8, 2, 64) for 32 Kbytes [MOTO93].

### 2.5.1 *Published Performance Data*

Most early published data on $P_{miss}$ is on unified caches (see [GIBS67, LIPT68, LEE69, MEAD70, KAPL73, BELL74, SMIT82, GOOD83]. Some of these papers report on cache performance, while in other papers $P_{miss}$ data is reported for various cache organizations and workloads. The published unified cache data has usually been secured by trace-driven simulation, and the results are usually reported for varying cache size, varying block size, varying degree of associativity, etc.

Smith [SMIT87] published $P_{miss}$ data from an investigation into cache and block size variations that segregated the $P_{miss}$ data for unified, instruction and data caches. The $P_{miss}$ data is presented as *design target miss ratios* (DTMR) "to provide for the computer system designer a set of numbers that he or she can use to estimate the performance impact of certain design choices." Smith's data, however, has some problems. The only organization parameters changed are cache size and block size. The other organization parameters are not reported in the paper; however, it is assumed that the organization is (S, 1, B, Byte). One of the interesting aspects of the experimental unified cache $P_{miss}$ data of [MEAD70] is that it is reasonably close to the design target miss ratios of [SMIT87].

Research of [GEE93] provides information on unified cache $P_{miss}$ for various organizations when executing the SPEC benchmarks. Table 2.16

| | Number of Sets | | | | | | |
|---|---|---|---|---|---|---|---|
| No. of Sec./Set | 64 | 128 | 256 | 512 | 1K | 2K | 4K |
| 1 | .151 | .114 | .086 | .056 | .039 | .029 | .020 |
| 2 | .092 | .069 | .047 | .031 | .023 | .016 | .011 |
| 4 | .062 | .044 | .029 | .022 | .015 | .010 | .007 |
| 8 | .041 | .028 | .021 | .015 | .010 | .008 | .005 |

Block size is 16 bytes.

**TABLE 2.16** Unified cache $P_{miss}$, SPEC benchmarks, overall.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000118

shows excerpted $P_{miss}$ for the benchmarks, a combination of floating point and integer, and a block size of 16 bytes.

A unified cache, as with instruction and data caches, shows that the size of the cache is paramount and the organization is of secondary importance. For example, an 8-K cache, organized as (64, 8, 16, 1), has a $P_{miss}$ of 0.041 while the organization (512, 1, 16, 1) has a $P_{miss}$ of 0.056. On the other hand, for cache design where the number of sets is constrained for some reason (discussed in Chapter 4), doubling the size of the cache by increasing the number of sectors can cause a significant decrease in $P_{miss}$. For example, an organization (256, 2, 16, 1) has $P_{miss}$ = 0.047 while (256, 4, 16, 1) has $P_{miss}$ = 0.029.

### 2.5.2 *Contemporary Unified Caches*

Many contemporary first-level caches are designed for VLSI implementation of new processors and new operating systems. For a number of reasons these caches are split as discussed in Section 2.5. All known new processors are implemented as pipelines that place heavy demands on the memory system. Because of this factor, some form of first-level split caches or multilevel caches is the design of choice today. The exception to the universal use of split caches is the IBM RS/6000 Model 220, which has a unified cache purely because of a marketing consideration; this processor model is targeted for a low-cost, low-performance market.

However, with split on-chip caches, a second-level unified cache is the norm. For example, the VAX 9000 has a first-level instruction cache followed by a second-level unified cache. The i486 has a second-level unified cache and an instruction queue. These caches are discussed in Section 2.6.

## 2.6 Split and Multilevel Caches

I can find no historical evidence of separate data and instruction caches before the mid-1980s. During the generation typified by IBM S/360 and VAX architectures, the caches were generally unified. It is only since microprocessor pipelining that the possibility of making concurrent references to instructions and data and with VLSI technology that designers have considered split and multilevel caches.

The major reason for using split caches is to provide a second path to memory from the processor. With pipelined processors, an instruction fetch and a data read/write can occur on the same clock, creating a structural hazard if there is only a unified cache. Two caches eliminate this hazard and permit concurrent references. Early use of two paths to memory were called "*Harvard Architectures*" [CRAG80]. Structural

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000119

hazards are discussed further in Chapter 8. However, a split cache has problems with self-modifying code because a specific instruction could have two values—one in the instruction cache and another value in the data cache.

Another reason for the use of split caches is that $P_{miss}$ is different for instructions and data. A look at the data from Tables 2.12, 2.13, 2.14, and 2.15 shows that significant differences between instruction $P_{miss}$, data $P_{miss}$, and unified $P_{miss}$ can exist for caches of equal size. For example, a 16-Kbyte cache organized as (512, 2, 1, 16) will have instruction $P_{miss} = 0.01$, data $P_{miss} = 0.104$, and unified $P_{miss} = 0.047$ for the overall SPEC benchmarks. With these differences, the possibility exists that the design of the three caches of a two-level cache system can be customized. There is a significant problem with split caches if self-modifying code is used, resulting in coherency problems between the two caches. A modified instruction in the data cache may be referenced in the instruction cache—a problem that must be managed either by software or hardware means.

Most of the early research on hierarchical memory systems recognized that there would be a continuum of levels [MATT70, MEAD70] and not just the processor/cache/memory levels. With practical VLSI and pipeline implementations, interest in multilevel caches has revived. Przybylski [PRZY90] discussed two reasons for renewed interest. First, all of the information presented previously shows that the $P_{miss}$ of a cache is not significantly improved if the cache size is increased beyond 64 Kbytes. Thus, ever larger caches are ineffective. Second, transport time can be reduced with a second-level cache, which reduces the number of cycles of a first-level cache miss. The reason for this is basically that the clock period for determining transfer time between the first- and second-level caches is much shorter than that between a single-level cache and the main memory.

Baer [BAER87] reports that the only system, as of 1987, to have a multilevel cache is the Facom M-3825. The first-level cache is organized as (64, 16, 1, 64) 64-Kbytes write through, while the second level is organized as (1024, 4, 1, 64) 256 Kbytes write back. Note that contemporary multilevel cache systems usually employ a very large second-level cache. The reason for this is that with VLSI memory devices, the second-level cache can be made large enough to hold a complete working set, thereby significantly reducing bus traffic to the main memory.

As shown in Figure 2.1, there are a number of topological configurations of cache systems that cannot be simply classified. This section will describe some configurations that have been employed. A partial listing of processors and their topologies is given in Table 2.17. Examples of the cache designs are described in the paragraphs below.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000120

| First Level | Second Level | Examples |
|---|---|---|
| I-Queue | Unified | i386, AM29000 |
| I-Queue | Unified, off-chip | i486, PowerPC 601 |
| I + D Unified | | |
| I-Cache | Unified, off-chip | MC68030, MC68040 |
| D-Cache | | MC88200, RS/6000 |
| I-Queue | Unified | VAX 9000 |
| I-Cache | | |
| I-Queue | Unified, off-chip | Intel Pentium |
| D-Cache | | |

**TABLE 2.17** Processor cache topologies.

First level, I-Queue;
Second level, Unified Cache;
Example, i386

The i386 processor is implemented with limited chip area with space for only an instruction queue. The supply of instructions to the queue and the supply of data are provided by an off-chip unified cache [ELAY85]; the organization is shown in Figure 2.23.

As previously discussed in Section 2.3.2, the on-chip instruction queue is used to smooth out the flow of variable-length instructions to the processor and hide cache misses. The off-chip unified cache organization and size are the responsibilities of the system designer for this processor. However, a suggested cache design is given by Intel in

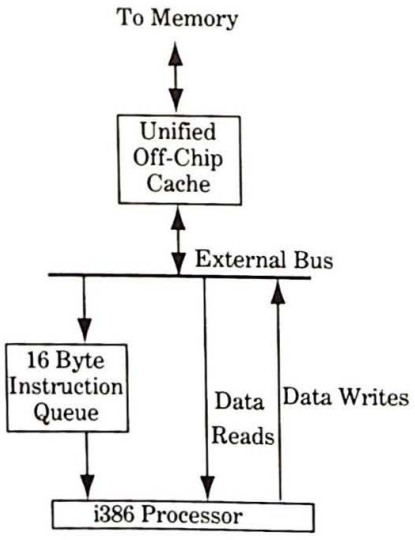



**FIGURE 2.23** i386 cache and instruction queue.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000121

[INTE86] along with an application note on the design of caches in general [SHIR86].

One of the major considerations that led to the selection of a unified cache is that the i386 had to execute old versions of MS-DOS that have self-modifying code. This consideration dictated a unified cache, external to the processor, for the i386. Intel also provides a cache component (part number 82396SX) for use with the i386. This cache organization is (256, 4, 1, 16) using a pseudo-LRU replacement algorithm and is the precursor to the i486 on-chip data cache, which is discussed below. Shires [SHIR86] provides hit ratio and performance ratios of various i386 unified cache organizations, executing an anticipated workload. This data indicates the dominant effect of cache size rather than organization as noted in previous sections.

First level, I-Queue;
Second level, Unified Cache;
Example, AMD 29000

AMD moved away from split caches for the AM29000 family. Software portability problems, in addition to complex bus issues, led to a return to a single bus access to the main memory. There is a four-word instruction queue (called a prefetch buffer by AMD) and a very large (192-word) register file that provide cache-like characteristics. A single bus services requests to memory that may be partitioned into instruction and data regions [AMD90].

First level, I-Queue + Unified;
Second level, Unified Cache;
Example, i486

The designers of the Intel i486, [FU89, CRAW90], needed to support t. memory accesses (instruction and data) per clock period. The instruction length is 1 byte to 15 bytes and the data word is 1 byte to 8 bytes. Thus the cache bandwidth needs to be in the range of 2–23 bytes per clock. In addition, the designers had a chip budget of 8 Kbytes in which to provide an on-chip cache.

As with the i386 cache, a unified cache is needed to support old MS-DOS and OS/2 systems that did not have "cache flush" instructions that evict the complete cache. These instructions are generally needed to support cache consistency. See Section 2.8.1 for more details.

The organization of the i486 cache is (128, 4, 1, 16) [INTE92]. To provide the ability to read instructions and read/write data simultaneously, the i486 has an instruction queue, as shown in Figure 2.24 and described in Section 2.3.2. The size of the cache and the number of sets are directly related to the size of the page (4 Kbytes) used in the virtual

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000122

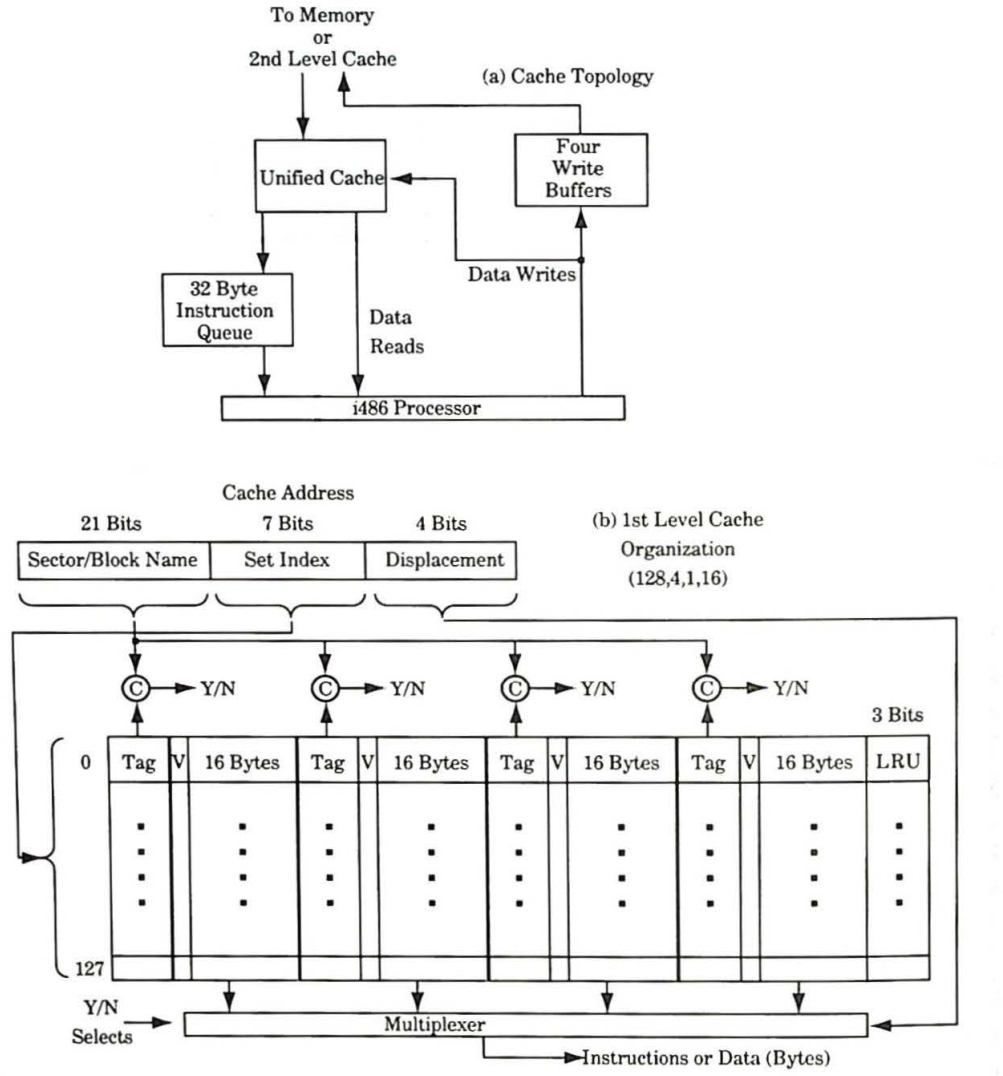



**FIGURE 2.24** Intel 486 cache.

memory. By matching these dimensions, one-half of the cache (two sector-blocks from all of the sets) is equal to one page. This organization permits a cache access and an access to the TLB to be made in parallel (an issue that is discussed in Chapter 4).

Each of the 128 sets has four sector-blocks with tags and valid bits. Because the write policy is write through, dirty bits are not required. However, each set has three bits (B2, B1, B0) for maintaining a pseudo-random LRU status of the sector-blocks within each set. These bits are set/reset as the cache is filled and sector-blocks are modified as shown in Table 2.18.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000123

| Sector-Block Used | B2 | B1 | B0 |
|---|---|---|---|
| S0 | X | 1 | 1 |
| S1 | X | 0 | 1 |
| S2 | 1 | X | 0 |
| S3 | 0 | X | 0 |

**TABLE 2.18** i486 cache LRU bits.

If all of the valid bits are "set," a sector is selected for replacement using the LRU bits that have recorded the use patterns of the sectors. Selection is simply the decoding of the B bits. For example, [0X0] identifies S3 as the LRU sector-block, and it should be overwritten; write back is not required since the write policy is write through.

A complete discussion of the design tradeoffs of the i486 cache is found in [GROC89]. The i486 cache design was tested by trace-driven simulation and gave a $P_{miss} = 0.08$ versus the $P_{miss} = 0.044$ shown in Table 2.16 for the same organization executing the SPEC benchmarks. These simulations showed that with a fast bus, the write through policy is only slightly faster than a write back policy. As the bus slows down, the write through cache slows down more than the write back and the bus utilization increases more for the write through cache. However, a write through cache is simpler to implement and was selected for the i486. With a write through policy, the bus can be expected to be utilized approximately 68% of the time, making shared memory multiprocessor configurations difficult to implement.

The instruction queue prefetches instructions via a 16-byte-wide bus from the unified cache in one clock into the prefetch queues. Thus, one block, which is four average-length instructions, is fetched into the queue in one clock. Instructions are then read from the queue into the processor.

It is interesting to note that, because the cache is on the same chip as the processor, the 16-byte bus between the cache and the prefetch queues poses minimum wiring problems as would be the case if such a wide bus is required to traverse chip boundaries.

First level, I-Queue + D-Cache;
Second level, Unified Cache;
Example, MC68030

The Motorola MC68030 has on-chip split instruction and data caches. Due to the limited chip area, these caches are only 256 bytes each. Because these caches are small, an effective application of this processor demands an off-chip second-level unified cache. Figure 2.25 illustrates the cache topology and the identical instruction and data caches.

The instruction and data caches are organized identically as (16, 1,

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000124

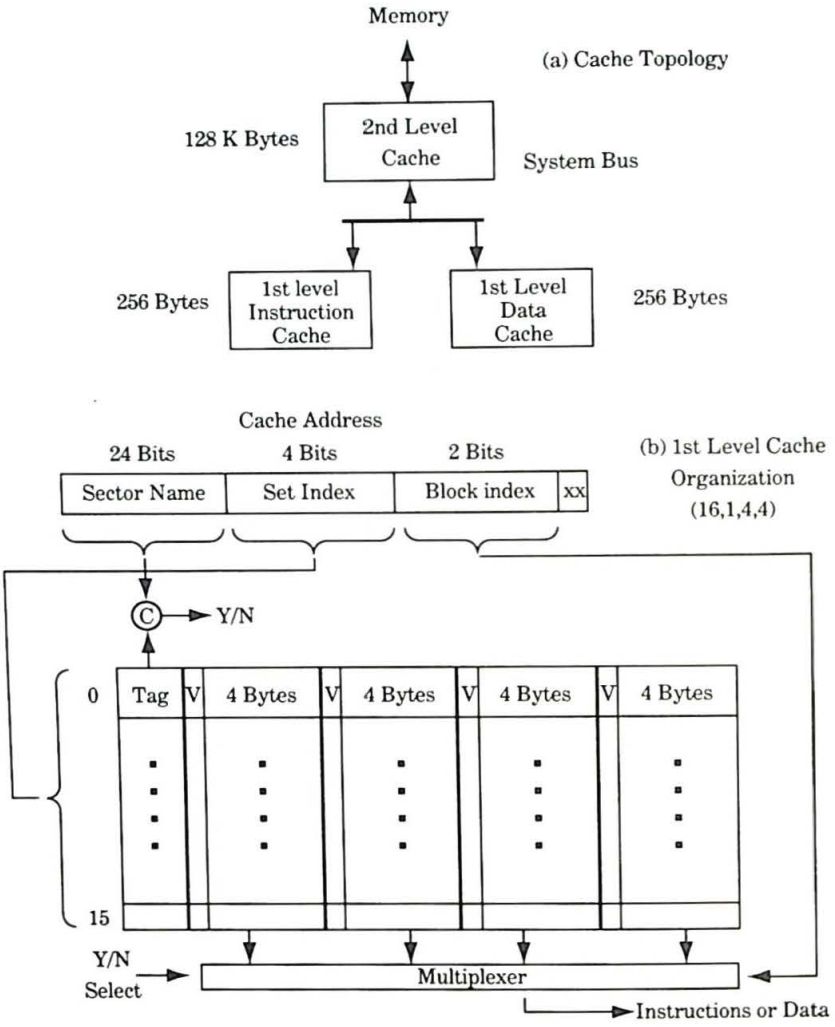



**FIGURE 2.25** Motorola 68030 instruction and data caches.

4, 4); the set index of four bits selects one of the 16 sets. If the set is present, the two block index bits select one of the four 4-byte words in the block; each block has a valid bit. The data cache uses a write through policy on a write hit. The data cache policy on a write miss is program-selectable for either write-allocate write back or no-write-allocate write through. As with all instruction caches, there is no write policy for the instruction cache. Furthermore, as self-modifying code is not supported, the instruction read process does not have to check the data cache for an instruction that has been modified.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000125

First level, I-Queue + D-Cache;
Second level, Unified Cache;
Example, DEC Alpha

The DEC Alpha processor, introduced in 1992, has a split cache on-chip and a second-level cache off-chip. The data cache has a capacity of 8 Kbytes, write through, direct mapped, and write-allocate. The data cache is byte addressable and the organization is (256, 1, 1, 32). The instruction cache has a capacity of 8 Kbytes, is also direct mapped with a 32-byte block, and is organized (256, 1, 1, 32). Both caches are addressed with the real addresses. The second-level cache, named the B-Cache, is unified, and the design is the system designer's responsibility.

First level, I-Queue + I-Cache;
Second level, Unified Cache;
Example, VAX 9000

The VAX 9000 [FITE90] has a multilevel cache consisting of a first-level instruction cache of 8 Kbytes and a second-level unified cache of 128 Kbytes as shown in Figure 2.26. There is no first-level data cache; data transfers directly to and from the register file and the second-level cache. The variable-length instructions are issued from a 25-byte instruction queue.

The instruction cache organization is (256, 1, 2, 16) and is addressed with virtual addresses to reduce latency and simplify the design. The cache address has a one-bit (B) block address and a one-bit (D) displacement address. As the cache AU is eight bytes, the three LSBs of the address are not used. A miss can load either one or two blocks as there is a valid bit per block. There are two sets of 256 valid bits (one for each sector-block); these bits are not integrated into the cache array (similar to the IBM S/360/85 cache).

The second-level unified cache is organized as (1024, 2, 1, 64) and is also addressed with virtual addresses. This cache is 2-way set associative and uses a FIFO block replacement scheme. It is believed that a write back policy is used in conjunction with a write queue.

First level, I-Queue + I-Cache + D-Cache;
Second level, Unified Cache;
Example, Intel Pentium

The recently announced Intel Pentium employs a split cache, unlik previous members of the ix86 family. The very large number of transistors available to the designers allowed them to solve the self-modifying code problem. The instruction read path is, starting at the processor, via a decoded instruction queue, an instruction queue, an instruction cache,

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000126

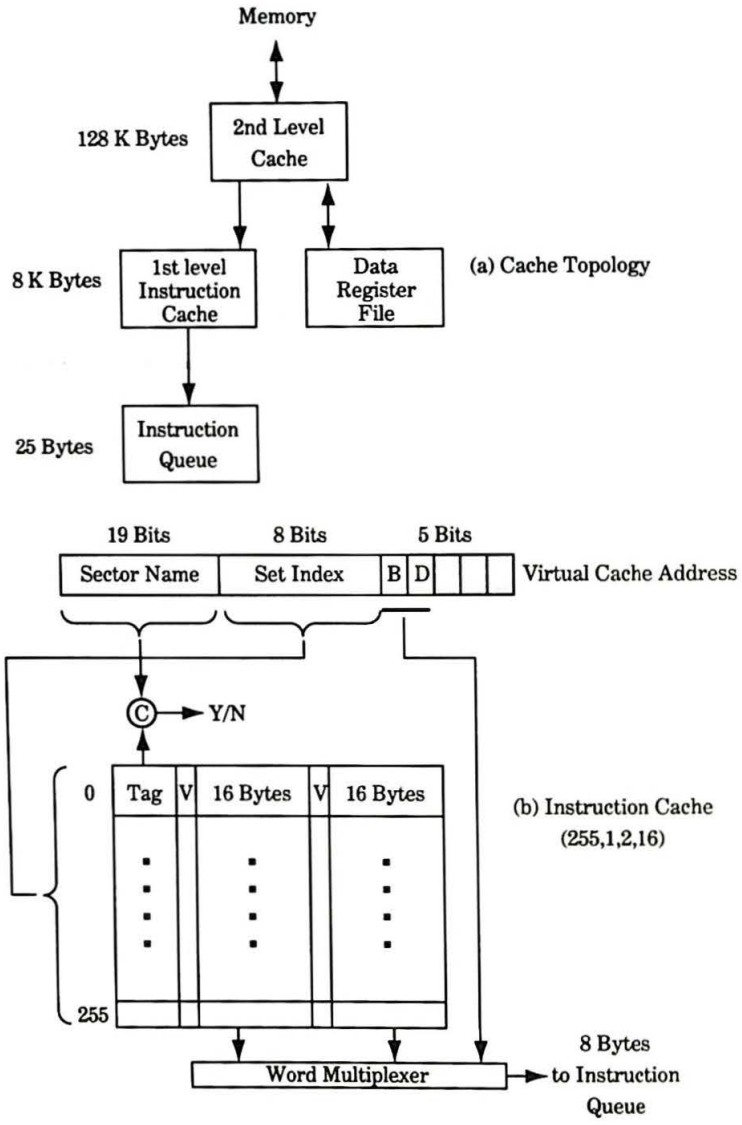



**FIGURE 2.26** VAX 9000 cache.

a unified cache, and the main memory. The data read path is via a data cache, unified cache, and the main memory.

As discussed previously, the Intel ix86 family of microprocessors have employed unified caches because of the problem of self-modifying code. This design philosophy has changed with the Pentium processor [INTE93, CRAW92, GWEN92, SLAT92]. There are two on-chip caches, an 8-Kbyte instruction cache and an 8-Kbyte data cache, and both are organized as (128, 2, 8, 4). The transfer path between the caches and the main memory is via a 64-bit bus. Logic is included to test for true

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000127

dependencies in the data cache to determine if instructions have been modified. Dependency management is discussed in Chapter 8.

The instruction queue can supply up to three instructions in one clock, while the data cache can supply two 32-bit words to the register file. The data cache has a unique interleaved organization that supplies operands to the superscalar pipelines (this design is described in Section 10.2.2).

## 2.7 Multilevel Caches

The idea at work with a multilevel cache is that a small and fast first level cache is used and that the second-level cache can (1) reduce the transport time to the first-level cache and (2) reduce the bus loading. With the proper interplay of these ideas, the cache system performance will be improved. Reduction in transport time is advantageous because on-chip caches were initially small with a resulting high $P_{\text{miss}}$. Because the weighted time to reference an AU not in the cache is the product of $P_{\text{miss}}$ and transport time, these two parameters are kept in balance with a second-level cache. (Another form of multilevel cache, *disk cache,* is discussed in Section 4.2.4.) Bus-loading reduction is important for multiprocessor systems where a number of processors must communicate over one bus to the main memory.

A simple two-level unified cache system is shown in Figure 2.27. This system has caches L1 and L2 between the processor and the main memory. The access time of L1 is $t_1$, the transport times for the second-level cache and the main memory are $t_2$ and $t_3$, respectively; and $P_{\text{miss1}}$ and $P_{\text{miss2}}$ are the *local* $P_{\text{miss}}$ for the two caches. *Local* $P_{miss}$ is the number of misses in a specific cache divided by the number of references to that cache.

The effective access time, as seen by the processor, for the simple two-level cache system using the sequential-forward model and local $P_{\text{miss}}$ values is

$$t_{\text{ea}}(\text{two levels}) = t_1 + P_{\text{miss1}} \times t_2 + P_{\text{miss1}} \times P_{\text{miss2}} \times t_3 .$$

The performance of a two-level cache can be compared to a one-level

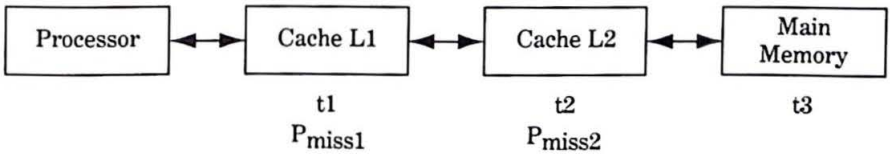



**FIGURE 2.27** Two-level cache.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000128

cache under the assumption that the $P_{miss}$ of a single-level cache is the same as the $P_{miss}$ of the first-level cache C1. The speedup is

$$S = \frac{t_{ea}(\text{one level})}{t_{ea}(\text{two levels})} = \frac{t_1 + t_{ea} P_{miss} t_3}{t_1 + P_{miss1} t_2 + P_{miss1} P_{miss2} t_3}.$$

For example, assume the parameters

$$\begin{aligned} t_1 &\quad 1, \\ t_2 &\quad 4, \\ t_3 &\quad 10, \\ P_{miss1} &\quad 0.05, \\ P_{miss2} &\quad 0.02. \end{aligned}$$

Using these parameters, the speedup of the two-level cache over the single-level cache is $S = 1.5/1.21 = 1.24$. In other words, adding this second-level cache provides a reduction of approximately 20% in effective access time. Note that $P_{miss}$ for different levels are not independent. Changing the design of one level will, in many caches, influence the $P_{miss}$ of another level. In the context of multilevel caches, two other definitions are of interest:

*Global* $P_{miss}$ is the number of misses in a cache or memory in the system divided by the total number of references to the memory system from the processor.

*Global* $P_{hit}$ is the number of references satisfied in a cache or memory in the system divided by the total number of references to the memory system from the processor. The sum of all global $P_{hit}$ values of a system is 1.

For an example of global $P_{miss}$ values and global $P_{hit}$ values, consider the following. If local $P_{miss1} = 0.1$ and local $P_{miss2} = 0.05$, then the *global miss ratio* for level two is $0.1 \times 0.05 = 0.005$. A more useful global metric is the global hit ratio. In this example, 90% of all references are satisfied in the first-level cache, 9.5% in the second-level cache, and 0.5% in the main memory.

The complexity of selecting an organization and policy for a one-level cache is compounded in the case of a two-level cache. The number of points in the design space is approximately the square of the number of design points of a one-level cache. For example, what is the policy when there is a miss on one of the caches? And, what is the policy when there is a miss on both of the caches. J.-L. Baer [BAER87] provides a discussion on these design problems.

To illustrate the complexity of a policy for a two-level cache, consider that the four write policy designs discussed in Section 2.2.6 will expand

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000129

| | L1 | L2 | ACTION |
|---|---|---|---|
| READ | Hit | X | read L1 → processor |
| * | Miss | Hit | Reload L2 → L1, read L1 → processor |
| | Miss | Miss | Reload Mem → L2, L2 → 1, read L1 → Processor |
| WRITE | Hit | Hit | Write processor → L1 → L2 |
| * | Hit | Miss | Allocate to L2, write processor → L1 → L2 |
| * | Miss | Hit | Write processor → L2 |
| | Miss | Miss | Allocate to L2, write processor → L2 |

*Does not provide multilevel inclusion.

**TABLE 2.19** Multilevel cache policies.

to sixteen possible designs for a two-level cache. Not all of these des.. are reasonable. One example that illustrates a particular selection of write strategies is found in the policy described in [BAER 87] and in Table 2.19. The write policies of the two caches are

Level 1: (hit) Write Through, (miss) No Write-allocate (Design A);
Level 2: (hit) Write Back, (miss) Write-allocate (Design C).

With this multilevel cache if, for example, there is a miss on L1 and a hit on L2, the block in L1 is replaced by the block in L2 and the new ALU is read into the processor. The actions in the table are viewed as logically sequential; however, a parallel implementation is possible. In other words, the new block of this example can be placed in L1 and forwarded to the processor.

For a single-level cache, the assumption is made that resident blocks in the cache must be found in the main memory. In other words, all blocks in the cache are *included* in the main memory. The cache system cannot have a valid block that is not represented in the main memory; the values may not be the same due to the delay waiting for a write back. With the addition of another cache level, *multilevel inclusion* (MLI), which means that the contents of a cache at level $l + 1$ is a superset of the contents of its cache at level $l$ (MATT70, LAM79, BAER87]), may not exist unless explicitly enforced.

The lack of MLI is illustrated by an example. Consider a block that has been moved to the L2 and L1 caches due to misses at both levels. A subsequent reference to the L2 cache (from I/O, say) may, due to a set conflict miss, evict that block. Now the block exists only in the L1 cache and in the main memory, not in the L2 cache. In this case,

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000130

multilevel inclusion is not provided. When there is a multiprocessor with a cache for each processor, MLI is more difficult to achieve.

Note in the example of Table 2.19 that this cache design does not provide MLI. For the read cases, as there can be an L1 hit and L2 miss, L2 is not a superset of L1. Likewise, as there can be a write with a miss on L1 and a hit on L2 and a hit on L1 with a miss on L2, L2 is not a superset of L1. Baer states that the first-level cache must be write through if multilevel inclusion properties are to be maintained.

There is also the issue of addresses presented to the various cache levels. The question of whether a cache should be addressed with real or virtual addresses will be discussed in Chapter 4. With a multilevel cache the issue is even more difficult. In fact, [WANG89] argues that the first-level cache should be addressed with virtual and the second-level with real addresses, an organization named *V-R caching*.

### 2.7.1 *Published Performance Data*

Przybylski [PRZY88] provides a wealth of information on the design and optimization of multilevel cache. Data on the effect of a second-level cache on bus utilization and increases in system performance are found in [GOLB92] for i486 processors. Bus utilization reduction is compared to an i486DX processor without a second-level cache. Two bus configurations are evaluated, namely, a slow bus that has a one clock hit and a four clock miss time and a fast bus with a one clock hit and a four clock miss time. The second-level cache is set at 128 Kbytes, 256 Kbytes, and 512 Kbytes. Excerpted bus utilization data are shown in Table 2.20. Note that a large second-level cache reduces the bus utilization to a negligible amount.

The performance improvement with a second-level cache was also measured and normalized to a 25-MHz i486 without a second-level cache. A 50 MHz without a second-level cache was evaluated to show the performance improvement obtainable with raw processor performance, and then this system was measured with a 256-Kbyte second-level cache.

| Second-level Cache | Slow Bus | Fast Bus |
|---|---|---|
| None | 88% | 68% |
| 128 Kbytes | 19% | 11% |
| 256 Kbytes | 15% | 6% |
| 512 Kbytes | 5% | 4% |

**TABLE 2.20** i486 bus utilization with a two-level cache.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000131

| System | Benchmarks | | |
|---|---|---|---|
| | ispec | fspec | spec |
| 25-MHz i486 (none) | 1 | 1 | 1 |
| 50-MHz i486 (none) | 1.61 | 1.62 | 1.61 |
| 50-MHz i486 (256 Kbytes) | 2.09 | 2.1 | 2.1 |

**TABLE 2.21** i486 performance improvement with a second-level cache.

The results of measurement while executing three SPEC benchmarks are excerpted in Table 2.21.

The interesting results here are that while doubling the clock rate of the processor gives a speedup of 1.6, adding a second-level cache gives an additional speedup of 1.3. This result compares favorably with the results of the speedup model developed earlier.

## 2.8 Other Cache Considerations

In addition to the design issues of organization, policies, and performance, there are other issues that influence the design and efficacy of caches. Some of these issues are control of the cache by the processor, transport time reduction techniques, compiler optimization, and pipelined caches.

### 2.8.1 *Processor Control*

Caches have historically been viewed as transparent to the user or processor. That is, the cache provides lower latency reads and writes than can be provided by the main memory, and the executing program should not be aware of the presence of a cache. This view of a transparent cache is no longer valid, and it has been found that there are situations that require direct processor control of the cache. A cache that has processor control is known as a *software visible cache*.

Bakoglu discusses the requirements for software visible caches [BAKO90].

> Software visible caches have two implications: The first is that any program that uses data references to create instructions (for example, loaders, debuggers, and simulators) must explicitly force these instructions from the data cache into the instruction cache. Additionally, to perform input or output operations on the processor main

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000132

memory, device driver code must properly flush the necessary lines from the caches before the I/O operation can begin.

There are two types of processor control exercised over the cache. These are:

1. establishing the cache operation mode,
2. dynamic pre-emption of normal cache functions.

## Establishing Cache Operation Mode

As has been discussed previously, a processor may be used in an application that may favor a particular cache operation mode. For example, where bus loading is a factor, a write back mode is desired because the write through mode can load the bus with unnecessary traffic. However (as discussed in Chapter 4), data coherency maintenance may be desired, suggesting a write through cache. Consequently, a cache with an operational mode set by the designer is less desirable than a cache that can be set to a particular mode by the user of the system. The operating mode of a cache is usually established by a mode control register that is loaded by the processor. Thus, a move instruction is executed in order to change or modify the operation mode of the cache.

An example of processor control of a cache mode is found in the MC68040 [MOTO89]. Four modes can be specified.

1. *Noncacheable mode.* For various reasons, it is desirable to prevent some AUs from being cached. One example of this need is found in a multiprocessor system that has multiple caches. A noncacheable mode can be set by the processor to prevent some shared variables from being cached.

If all shared variables are cached, a coherency problem can exist hat can be solved by not caching. The processor can thus specify that che following references are not to be cached. Reads, and writes that miss, do not allocate but access the main memory. Noncached writes that hit a resident cached block cause the valid bit to be reset, and the block, if dirty, is written to the main memory before the write access is made to the main memory.

2. *Cacheable, write through mode.* In addition, for cached operation, the processor can specify the write policy to be write through. Write through is handled as a no-write-allocate policy.

3. *Cacheable, copyback (write back) mode.* For the cacheable write back mode, write hits update the cache and set the dirty bits. A write miss is handled by allocation.

4. *Special access.* The last mode is special access. Read or write accesses that miss the cache do not allocate but read or write to/from

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000133

the main memory. Read or write hits are handled by a mode specifier in the access address.

Another example of processor control over caches is found in the MC68030 [MOTO87]. The on-chip data cache can be set for either allocate or no-write-allocate on a write miss.

### Pre-emption of Cache Functions

Recall that the purpose of a cache is to reduce the latency of reads and writes. Depending upon the operation mode, when a read or write instruction is executed, various control bits are set or reset and blocks are moved to/from the main memory. All of these actions can be viewed as *side effects* of the LOAD and STORE instructions as they hit or miss the cache. For particular situations it is desirable to pre-empt these side effects and execute instructions that explicitly cause the side effects. Pre-emptive fetches and evictions of the cache are desirable instructions for synchronizing the memory in a multiprocessor configuration.

Some processors, such as the PowerPC 601 [MOTO93], have the ability to evict the cache contents or invalidate the cache. There are four cache instructions: data cache block store (*dcbst*), data cache block flush (*dcbf*), data cache block set to zero (*dcbz*), and instruction cache block invalidate (*icbi*). If the cache organization is write through, flushing is done by an instruction that turns off all of the valid bits. For write back caches, flushing must move the modified contents of the cache to the main memory. The net effect of complete eviction is that subsequent cache operations are similar to a cold start and all references are misses.

Multiprogramming a virtual memory system with a virtual cache and a write back policy requires the facility to evict dirty blocks (an issue discussed in Chapter 3). A write through cache requires only a valid bit, and complete or partial eviction is not required. The software prefetching cited and discussed in Section 2.8.2 is another example of processor control over the operation of a cache.

## 2.8.2 *Transport Time Hiding or Reduction Methods*

Hiding or reducing the transport time between the main memory and the cache is a major avenue for improving cache and processor performance. Multilevel caches (discussed in Section 2.6) are one method used to reduce transport time while instruction queues (discussed in Section 2.3.2) and data queues (discussed in Section 2.4.3) can hide transport time. Other available techniques include hardware-controlled prefetching, software prefetching, and special-purpose caches. These techniques are discussed in the following sections.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000134

## Hardware Controlled Prefetching

Various techniques have been use to prefetch instructions and data under hardware control. By this we mean that prefetch mechanisms are wired into the processor and perform their function without software interaction.

### *Instruction Prefetch*

Smith [SMIT78] presented early research on the benefits of prefetching instructions from disk to memory and memory to cache. His results indicate that prefetching is only effective for small fetch units such as 32-byte to 64-byte blocks and is not effective for pages of approximately 1 Kbyte to 2 Kbytes. His results also show that prefetching can be effective in increasing CPU speed by 10% to 25% when prefetching is used at the memory–cache level.

Smith [SMIT82] provides a description of the issues involved in prefetching either instructions or data and notes three prefetch strategies.

1. *Always prefetch.* For every access to $AU_i$, a prefetch to $AU_{i+1}$ is initiated.
2. *Tagged prefetch.* When a block is referenced for the first time, the next block is prefetched.
3. *Prefetch on misses.* This prefetch type implies that the block is greater than one AU.

One form of instruction prefetching is found in processors that have instruction queues; prefetching is initiated by monitoring the content of the queue. When there is a vacancy in the queue that can be filled by a reference to memory, a prefetch cycle is initiated (this operation is described in Section 2.3.2).

Another form of hardware-controlled instruction prefetch is found in systems that prefetch a branch instruction's target stream in tandem with continued fetching of the in-line instruction stream. These systems are described in Chapter 7.

### *Data Prefetching*

Hardware data prefetching has usually been accomplished in register file architectures with a delayed load strategy. Useful instructions are interposed between the load instruction and the instruction that will use the loaded AU (a strategy discussed in Chapter 8). Another form of data prefetching is found in processors such as the IBM S/360/91 [ANDE67] that decodes load instructions in the instruction fetch queue and begins the load before the instruction is released to the execution unit. Data

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000135

returned from the memory are stored in a queue that joins the execution pipeline at the proper point. Memory delays are hidden by the queue.

A significant problem with data prefetch is that the effective address is sometimes not known until after some registers have been adjusted; this is a true dependency on addresses and is called an *address generation interlock* (AGI). R. J. Eickemeyer [EICK93] suggests ways for predicting the effective address and gives measured data to show the effectiveness of data prefetching.

## Software Prefetching

When a referenced AU is not found in a cache, a cache miss results and the missed AU is fetched along with a number of AUs in contiguous memory locations (the block). Some programs have addressing patterns that are highly predictable, such as signal processing and graphics. Under these situations, why not have an instruction that will prefetch a block into the cache rather than wait on a miss? Prefetching of instructions or data into caches that are nonblocking is an interesting concept. *Software prefetching* [CALL91] is a program-initiated, nonblocking load of a cache block in anticipation of need.

Prefetching into a nonblocking cache will not halt execution as does a normal cache miss because the executing program is not in immediate need of the instruction or data AU. Recall that when a normal miss occurs, the processor must wait until the miss has been serviced. If a prefetch command is issued well in advance of the need for the AU, there will not be a miss when the AU is read during the normal instruction execution cycle.

An example of a data-prefetching instruction is the PowerPC 601 [MOTO93]. The instruction Data Cache Block Touch (dcbt) loads a block into the cache. In other words, this is a load instruction that loads a block into the cache rather than loading an AU into the register file.

Simulation results of [CALL91] indicate that approximately 33% of all software data prefetches are useful in that they accurately anticipate a future reference to the prefetched block. With this level of success, the number of cache misses is reduced because the data has already been prefetched. However, the reduction in $P_{\text{miss}}$ is not without cost. The use of a prefetch instruction increases the number of instructions that must be executed, hence adding to the length of the program. If the cache is blocking, all prefetches steal cycles from the cache and ineffective prefetches degrade the available performance of the cache.

A performance model is developed to investigate the effectiveness of software prefetching. Consider a unified blocking cache of the sequential-forward type. The model is based upon the parameters

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000136

$F_f$ = fraction of normal instructions that are preceded by a prefetch instruction,
$P_{eff}$ = probability that a prefetch has been effective and a future miss does not occur,
$P_{miss}$ = probability that a memory reference misses in the cache for the program without prefetching,
$L$ = number of clocks the cache is blocked due to a prefetch,
$T$ = transport time.

There are eight possible events, with each event having a delay. The effective access time of the cache will be 1 + weighted average delay. These events and delays are shown in Table 2.22.

The event 0 1 0 is a normal cache access with a miss and a delay of $1 + T$ cycles. For the case 1 1 0, there has been an ineffective prefetch: the delay is $1 + L + T$ because the executing instruction misses the cache requiring the normal delay of $T$ cycles and $L$ cycles to be lost due to the blockage of the cache during the prefetch. Notice that when a prefetch instruction is executed ($F_f = 1$), the cache loses a cycle for the instruction fetch plus the number of cycles to transfer the block $T$. From the event probabilities, the effective cache cycle time is

$$t_{ea}(\text{PF}) = 1 + P_{miss}(T(1 - P_{eff}) - P_{eff}) + \text{F}_f(1 + L).$$

The effective cache cycle model is optimistic, as a prefetched block may evict a block already in the cache because of a capacity or set conflict. For the cases when unnecessary eviction happens, the delay would be two transport times. Prefetching may increase the $P_{miss}$ of the executing program because a prefetched block may evict a block in the cache that

| Events | | | Delay | Comments |
|---|---|---|---|---|
| $F_f$ | $P_{miss}$ | $P_{eff}$ | | |
| 0 | 0 | 0 | 0 | |
| 0 | 0 | 1 | 0 | |
| 0 | 1 | 0 | $1 + T$ | Previous prefetch not effective |
| 0 | 1 | 1 | 0 | |
| 1 | 0 | 0 | $1 + L$ | Cache blocked for $L$ cycles |
| 1 | 0 | 1 | $1 + L$ | Cache blocked for $L$ cycles |
| 1 | 1 | 0 | $1 + L + 1 + T$ | Cache blocked for $L$ cycles |
| 1 | 1 | 1 | $1 + L$ | Cache blocked for $L$ cycles |

$F_f = 0$ means no prefetching.

**TABLE 2.22** Software prefetching delays.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000137

may need to be fetched again. The eviction problem decreases as the degree of associativity of the cache increases. The speedup of using software prefetching is found by dividing $1 + P_{miss}\ T$ by $t_{ac}(PF)$.

$$\text{Speedup} = \frac{1 + P_{miss}T}{1 + P_{miss}(T(1 - P_{eff}) - P_{eff}) + F_f(1 + L)}.$$

Break-even points for software prefetching compared to not prefetching are found by equating $t_{ac}(PF) = 1 + P_{miss}\ T$ and solving for $P_{eff}$, $F_f$, and $T$ according to the equations

$$P_{eff} = \frac{F_f(1 + L)}{P_{miss}(1 + T)},$$

$$F_f = \frac{P_{eff}P_{miss}(T + 1)}{1 + L},$$

$$T = \frac{F_f(1 + L)}{P_{eff}P_{miss}} - 1.$$

For example, for a system with $P_{eff} = 0.3$, $P_{miss} = 0.1$, $T = 3$, $L = 2$, and $P_{eff} = 0.3$, no more than 4% of the instructions of the original program can have an associated prefetch instruction or prefetching will lose performance.

For a system with nonblocking caches $L = 0$, assuming that the cache can queue up an unlimited number of requests. For $P_{eff} = 0.3$, $P_{miss} = 0.1$, and $F_f = 0.1$, the break-even value of $T$ is 9. Thus, for systems with $T$ greater than 10, software prefetching may be beneficial with a nonblocking cache. With a blocking cache with $L = 3$, $T$ must be larger than 40 in order to improve the performance with prefetching.

The speedup and break-even models indicate a major concern: Software prefetching is unlikely to improve the performance for most systems with blocking caches and small values of $T$. In fact, the likelihood is that performance will be lost.

A significant contributor to reduced speedup is the overhead associated with increasing the program length with prefetching, that is, the factor $F_f$. D. Callahan et al. [CALL91] report that if $F_f \approx 0.28$, prefetching is unlikely to reduce the time of program execution. There are situations that may significantly reduce $F_f$ and favor the use of software prefetching. Superscalar processors with nonblocking caches (described in Chapter 10) that issue more than one instruction per clock may be the best candidates for prefetching because the prefetch instruction can be introduced into an instruction slot that would otherwise be unused. As a result, for many programs, there is no increase in program length (that

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000138

is, $F_f = 0$) due to the prefetch instructions. And, with a nonblocking cache, the cycles are not lost while waiting for the fetch of a block from the main memory.

Another situation concerns disk caches. Notice that the value of $T$ plays an important part of the possible speedup; cache systems with small values of $T$, in general, do not benefit. However, due to the difficulty of providing low values of $T$, software prefetching may be helpful when high values of $T$ are necessary. Moreover, for disk systems with very high values of $T$, software prefetching is frequently employed. Some operating systems (as discussed in Chapter 3), can fetch instructions and data into main memory from the disk before they are needed.

[CALL91] discusses the various considerations of software prefetching. These considerations include the time interval between the prefetch and the actual need (known as *prefetch distance*), the influence of block size, the need to abort prefetches already in the cache, and the additional load placed on the bus. Prefetch distance must be large enough so that the prefetched data is in the cache before the explicit reference is made to the data. On the other hand, if the prefetch distance is too large, cache residency time increases and the effective size of the cache is reduced. Long residency can also increase bus traffic because prefetched data may not be used due to a change of program flow. This research reports that the cache hit ratio for a suit of six benchmarks improved from an average 72.7% to 99.4% with prefetching.

A. J. Smith [SMIT78] gives a detailed description of the cache of the Amdahl 470V/6 and the IBM S/360/168–3 along with implementation suggestions for prefetching. He projects that the inclusion of prefetching could improve system performance by 10% to 25%. The machine studied required 7 to 8 clocks per instruction and had transport time cost of 12 to 15 cycles. These results may not be representative of pipeline microprocessors with an instruction execution rate of approximately 1 clock per instruction. For pipelined processors, the effect of cache delays is far more damaging and small improvements in $P_{\text{miss}}$ can have a significant positive performance impact if the other factors do not overshadow the $P_{\text{miss}}$ improvement.

The DEC Alpha architecture provides a data prefetch instruction, named *FETCH*, that moves an aligned 512-byte block from a low level of the hierarchy to a higher level. Specific implementations of the Alpha architecture may or may not provide this instruction and, if provided, the block moved may be smaller or larger than 512 bytes. The order of prefetching within the block is unpredictable. The break-even models developed above clearly show that software prefetching is only useful for systems that have long transport times. This conclusion is confirmed by the DEC Alpha handbook, which indicates that prefetching is useful for transport times on the order of 100 cycles, that latencies on the order

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000139

of 10 cycles will not be helped, and that code scheduling is suggested for these small latencies. I assume that these comments on the Alpha are based on a nonblocking data cache. One would not expect to improve system performance with software prefetching without including a nonblocking cache.

The MC88100 has a mechanism named *instruction prefetching* [MOTO90]. However, this mechanism is not software prefetching as described above but Motorola's way of describing the pipeline that contains two stages for instruction fetching, a common approach in pipelined processors. With a two-stage instruction fetch pipeline, two program counter stages are required. These are named the *fetch instruction pointer* and the *next instruction pointer*. The requirement for these registers is described in Chapter 8.

## Special-Purpose Caches

Recall from Section 2.1.4 that nonblocking caches permit read operations to proceed out of order without blocking. Another approach to nonblocking is to provide two caches, similar to double buffering, that eliminate the structural hazard of a single blocking cache. Klaiber and Levy [KLAI91] proposed a dual cache scheme that improves the chances of useful speedup. With this scheme, a blocking cache that is blocked during a block prefetch is augmented with a buffer that is used for storing the prefetched blocks. Subsequent reads to the cache search both cache and the buffer in parallel. Figure 2.28 shows a block diagram of this cache scheme.

A prefetch request is queued up to the memory that then places the requested block into the prefetch buffer. The fetch queue issues a fetch

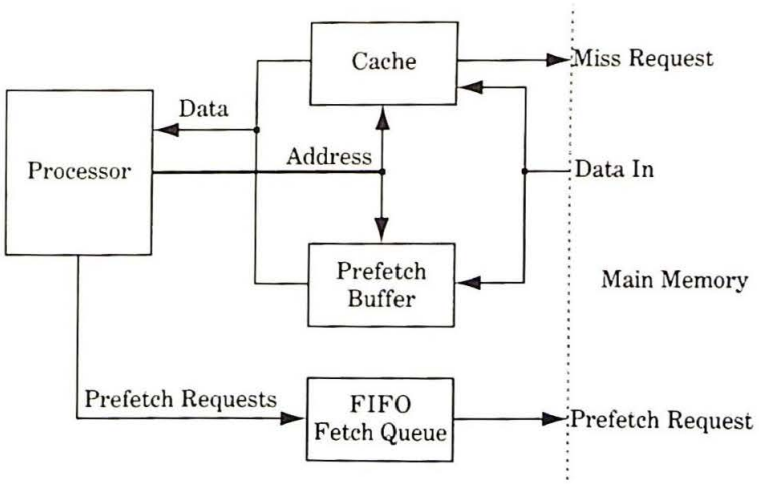



**FIGURE 2.28** Klaiber-Levy cache.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000140

request only if a miss from the cache will not be blocked. A read request checks both the cache and the prefetch buffer and takes an instruction or data from the one that hits.

With the prefetch buffer, there is no blocking of the main cache during the prefetch. The model for this cache is found by setting $L$ in the $t_{ac}$(PF) model to 0. The speedup becomes

$$\text{Speedup}(K - L) = (1 + P_{miss}T)/(1 + P_{miss}T(1 - P_{eff}) + F_f) .$$

Simulations using the Livermore Loops by Klaiber indicate that a memory speedup of 3 to 8 is achievable using this cache. For the model above and using the values $P_{miss} = 0.05$, $T = 13$, $P_{eff} = 0.9$, and $F_f = 0.05$, a speedup of 5.84 is predicted. This model prediction compares favorably to the speedup of 6.5 obtained from Klaiber's simulation.

Jouppi [JOUP90] proposes three augmented cache schemes that give the advantage of a small transport time. The idea behind the Jouppi extension is primarily the reduction of delays due to conflict misses. M. D. Hill [HILL87] shows that approximately 25% of all misses are conflict misses with a unified direct cache. If these 25% of the misses could have a transport time of one cycle rather than the normal 4- to 10-cycle transport time, then a significant improvement in cache performance is obtained. The idea is to cache the miss path from the main cache. Then the auxiliary cache is addressed in parallel with the normal direct cache. Jouppi's three schemes are illustrated in Figure 2.29.

1. *Miss cache.* A small, 2- to 5-word, associative cache is loaded with the AU from a miss in parallel with the AU being loaded into the direct cache. The two caches are accessed in parallel. If the block is subsequently evicted from the main direct cache, a one-cycle latency brings the missed AU into the cache as well as the processor. Miss caches are more effective for a data cache than for an instruction cache because data references are more random than instruction references.

2. *Victim cache.* A small, associative cache is loaded with the block that is evicted from the main cache due to a conflict miss. A future reference may find the evicted block in the victim cache; a one-cycle latency will bring the evicted AU into the processor. Jouppi claims that a victim cache will always outperform a miss cache.

3. *Stream buffer.* A small FIFO queue that can be associatively addressed in parallel with the main cache is provided. The stream buffer prefetches the addresses following the missed address of main cache. All references check the tags of both the main cache and the stream buffer and read the one that hits. Stream buffers reduce compulsory misses as well as capacity misses. Multiple-path stream buffers are also proposed.

The use of victim caches or stream buffers of 5 to 16 words reduces the apparent $P_{miss}$ of a first-level direct cache by a factor of two to

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000141

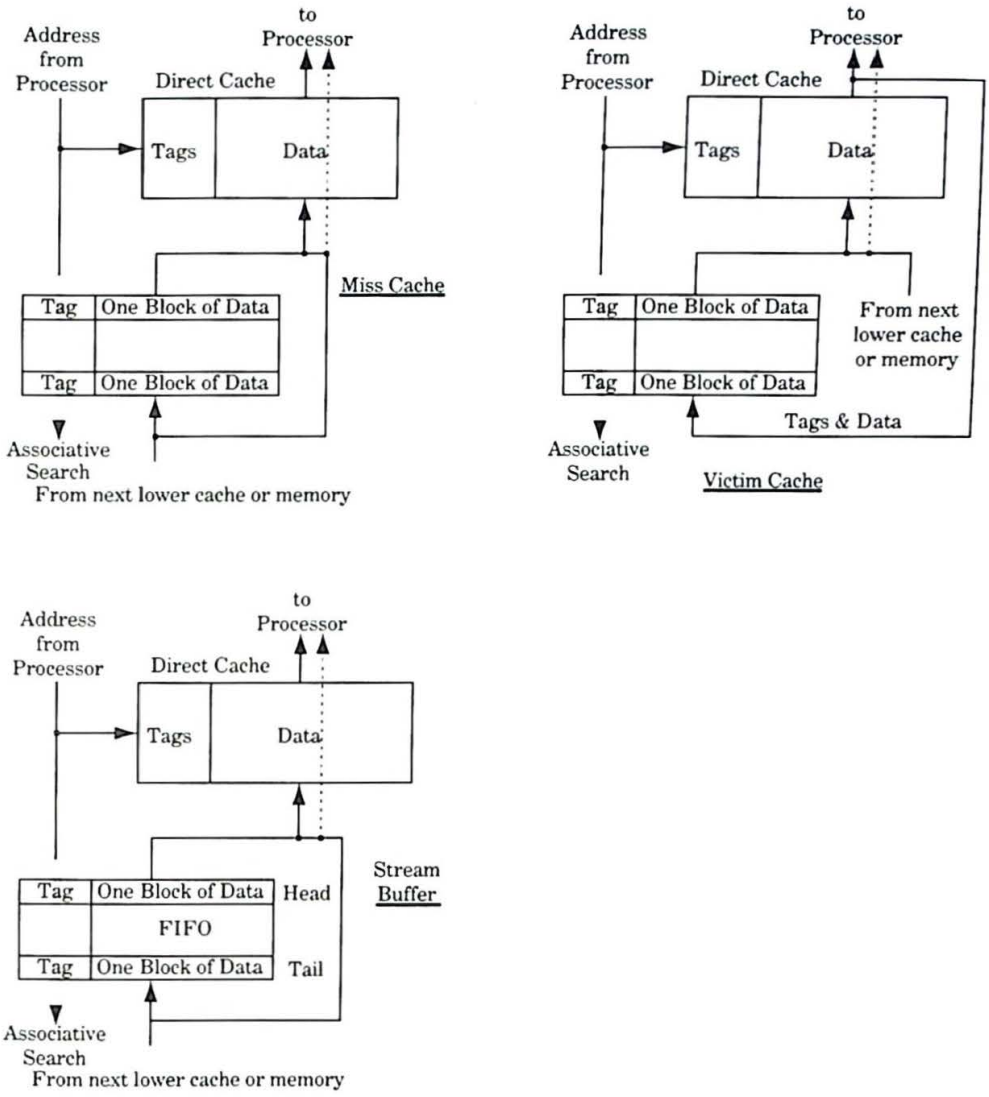



**FIGURE 2.29** Jouppi caches.

three depending upon the executing program [JOUP90]. For a pipelined processor with CPI = 1, a performance improvement of 15% to 25% is realized with a modest hardware investment.

Schauser et al. [SCHA91] describe a special instruction cache named a "stall cache." This cache is associated with processors having a single bus to the main memory. With a single cache, the designer has options such as (1) an instruction buffer, queue, or cache, and (2) an on-chip split cache. The stall cache is another design option for this problem. With this option, a split cache is off-chip and the stall cache is on-chip, as shown in Figure 2.30.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000142

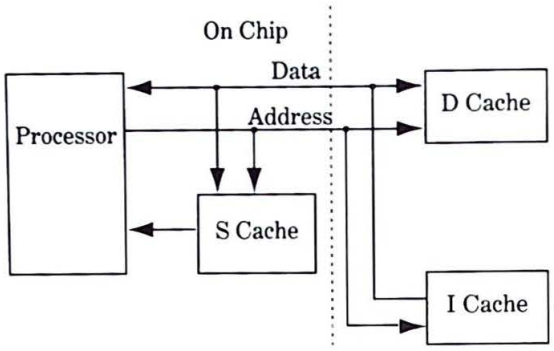



**FIGURE 2.30** Stall cache.

The instruction stream is usually fetched over the bus from the instruction cache. If a data read occurs at the same time as the instruction read, the instruction is taken from the stall cache. A miss on the stall cache is filled from the instruction cache. Consequently, the instruction path is similar to a two-level cache. Table 2.23 shows the events and actions for the stall cache.

For example, if there is a data read or write cycle and no instruction cycle, the read or write is from/to the D Cache. The stall cache is a restricted instruction cache, similar to a branch target cache (discussed in Chapter 7).

A simulation of the stall cache shows that the performance of the stall cache approaches the pure split cache with two buses. Equal performance requires an on-chip instruction cache that is 1.5 to 2 times larger than the stall cache. This evaluation assumes that the latency of an off-chip cache is the same as an on-chip cache (an assumption that may not be reasonable). Thus, the benefits of this scheme are to reduce the chip area needed for an instruction and/or split cache while retaining the single bus between the processor and the main memory. For example, if

| External Bus Cycles | | Instruction in Stall Cache? | Action |
|---|---|---|---|
| Inst. | Data | | |
| Yes | No | X | Read I Cache |
| No | Yes | X | Read D Cache |
| Yes | Yes | Yes | Read I from S Cache and D from D Cache |
| Yes | Yes | No | Read D Cache and add cycle to fetch instruction block into S Cache and processor |

**TABLE 2.23** Stall cache events and actions.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000143

a split cache requires 30% of the chip area, equally divided between the instruction and data caches, the chip area can be reduced by 20% if there is a 1.5 factor of instruction cache area reduction.

### 2.8.3 *Compiler Optimization*

The organization and policies of caches are selected to match the design to the expected address sequences of instruction and/or data. The measures of success are the reduction of effective access time, reduction of $P_{\text{miss}}$, and the reduction in the bus traffic.

Another approach to achieving the desired goals stated above is to use the compiler to reorder the instruction and data sequences so that, for a given cache design, the desired goal of reducing $P_{\text{miss}}$ can be achieved [HWU89, MCFA89]. The compiler attempts to maximize the temporal and spatial localities and minimize mapping conflicts.

The results of this research into compiler optimization show that, with compiler optimization, a direct cache compares favorably with a set associative cache of equal size. McFarling [MCFA 89] studied instruction caches only. His results show that compiler optimization can be as effective as increasing the cache size threefold. And, "Optimized programs on a direct mapped cache have lower miss ratios than unoptimized programs on set associative caches of the same size."

With the complexity of $n$-way set associative caches, the use of optimization should be exploited. In addition, the logic path of a direct cache is slightly shorter than that of an $n$-way set associative cache, thereby reducing the basic hit time of the cache.

### 2.8.4 *Pipelined Caches*

The assumption has been made in this chapter that the cache access time with a hit is a one clock cycle per operation. With the increasing demand for cache bandwidth, pipelining the cache is becoming an important design option. While pipelining is discussed in Chapters 6 to 9, the idea behind cache pipelining for performance reasons is briefly discussed here.

An early implementation of a pipelined cache is found on the Xerox Dorado [CLAR81]. This cache is an early select design that lends itself to pipelining, as shown in Figure 2.11. The first pipeline stage accesses the tag memory with the translation addresses and performs the tag check. The second stage accesses the data memory. The Dorado cache could support one cache reference per cycle, as long as there are no cache misses.

Recall, however, that modern caches are late select and can also be

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000144

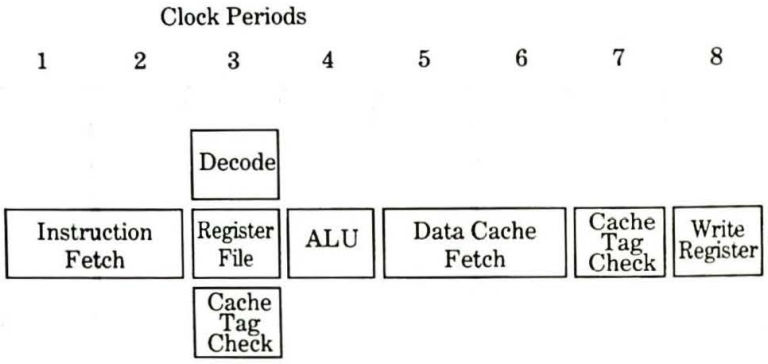



**FIGURE 2.31** R4000 pipelined cache.

pipelined into at least two stages: Stage (1) reading the cache and the tags and Stage (2) comparing the tags with the address. Because reading the data and tag memories is a longer operation than the tag check and selection operation, the two stage times allocated to these two steps are not balanced. With pipelining, the first stage can be decomposed into two stages, giving a three-stage pipeline with reasonable balance between the stage times. Note that an interstage register is not required between the stages (a form of maximum rate pipelines to be discussed in Chapter 6). Thus, reading the cache can be a three-pipeline stage operation, as illustrated by the MIPS R4000 [MIRA92] and shown in Figure 2.31.

With the MIPS R4000, an instruction is decoded and the register files are read in parallel with tag checking. Thus, decoding and register file reads are conditional and can be abandoned if the tags check fails because of an instruction cache miss. An incorrect tag check will abort the instruction before it is issued to the ALU. However, tag checking of a write to the register file that results from a LOAD instruction is performed in serial. The reason for serial checking is that an incorrect conditional write would require extensive buffering and load true dependencies can be resolved with short circuiting. These issues are discussed further in Chapter 7.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000145

# 3

# Virtual Memory

## 3.0 Overview

Virtual memory has evolved in response to a number of factors. First, the address space of early processors was closely matched to the size of the *real memory* (main memory) and, for many reasons, was found to be inadequate. Second, software schemes that enabled the operating system to manage the transfer of information from bulk storage were developed. Third, virtual memory helped to overcome the performance penalty generated by these software memory management schemes.

The organization, or structure, of the virtual memory system results from matching the data structures employed by programmers. Program (code and data) modularity has been an accepted practice for many decades to facilitate independent development and checkout of software. Modules can be viewed as a hierarchy, and the allocation system should facilitate the modular view. A common data structure for representing modules is the tree, as shown in Figure 3.1, that reflects the hierarchical nature of many commonly occurring data structures. Files are referenced from the Root Directory via interior nodes or subdirectories. The files for early versions of UNIX were variable-length segments.

It is informative to survey the evolution of software memory management systems that preceded virtual memory. Figure 3.2(a) shows that the early *simple batch monitors* provided a single contiguous memory partition for a single user. When the user program is changed—due perhaps to the termination of a job—the active user partition is written out to disk and a new job is read in. The major drawback with this system is the overhead incurred in swapping and the inefficient use of processor and memory resources.

These early batch monitor systems were also called *roll-in/roll-out* because of the way jobs are moved from disk to the main memory. Upon the first instance of the job being moved to the main memory, real addresses are assigned that are used in subsequent references. This



Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000146

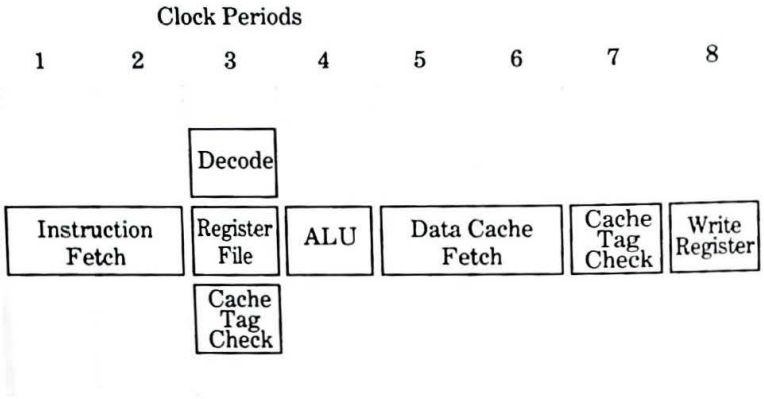



**FIGURE 2.31** R4000 pipelined cache.

pipelined into at least two stages: Stage (1) reading the cache and the tags and Stage (2) comparing the tags with the address. Because reading the data and tag memories is a longer operation than the tag check and selection operation, the two stage times allocated to these two steps are not balanced. With pipelining, the first stage can be decomposed into two stages, giving a three-stage pipeline with reasonable balance between the stage times. Note that an interstage register is not required between the stages (a form of maximum rate pipelines to be discussed in Chapter 6). Thus, reading the cache can be a three-pipeline stage operation, as illustrated by the MIPS R4000 [MIRA92] and shown in Figure 2.31.

With the MIPS R4000, an instruction is decoded and the register files are read in parallel with tag checking. Thus, decoding and register file reads are conditional and can be abandoned if the tags check fails because of an instruction cache miss. An incorrect tag check will abort the instruction before it is issued to the ALU. However, tag checking of a write to the register file that results from a LOAD instruction is performed in serial. The reason for serial checking is that an incorrect conditional write would require extensive buffering and load true dependencies can be resolved with short circuiting. These issues are discussed further in Chapter 7.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000147

# 3

# Virtual Memory

## 3.0 Overview

Virtual memory has evolved in response to a number of factors. First, the address space of early processors was closely matched to the size of the *real memory* (main memory) and, for many reasons, was found to be inadequate. Second, software schemes that enabled the operating system to manage the transfer of information from bulk storage were developed. Third, virtual memory helped to overcome the performance penalty generated by these software memory management schemes.

The organization, or structure, of the virtual memory system results from matching the data structures employed by programmers. Program (code and data) modularity has been an accepted practice for many decades to facilitate independent development and checkout of software. Modules can be viewed as a hierarchy, and the allocation system should facilitate the modular view. A common data structure for representing modules is the tree, as shown in Figure 3.1, that reflects the hierarchical nature of many commonly occurring data structures. Files are referenced from the Root Directory via interior nodes or subdirectories. The files for early versions of UNIX were variable-length segments.

It is informative to survey the evolution of software memory management systems that preceded virtual memory. Figure 3.2(a) shows that the early *simple batch monitors* provided a single contiguous memory partition for a single user. When the user program is changed—due perhaps to the termination of a job—the active user partition is written out to disk and a new job is read in. The major drawback with this system is the overhead incurred in swapping and the inefficient use of processor and memory resources.

These early batch monitor systems were also called *roll-in/roll-out* because of the way jobs are moved from disk to the main memory. Upon the first instance of the job being moved to the main memory, real addresses are assigned that are used in subsequent references. This



Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000148

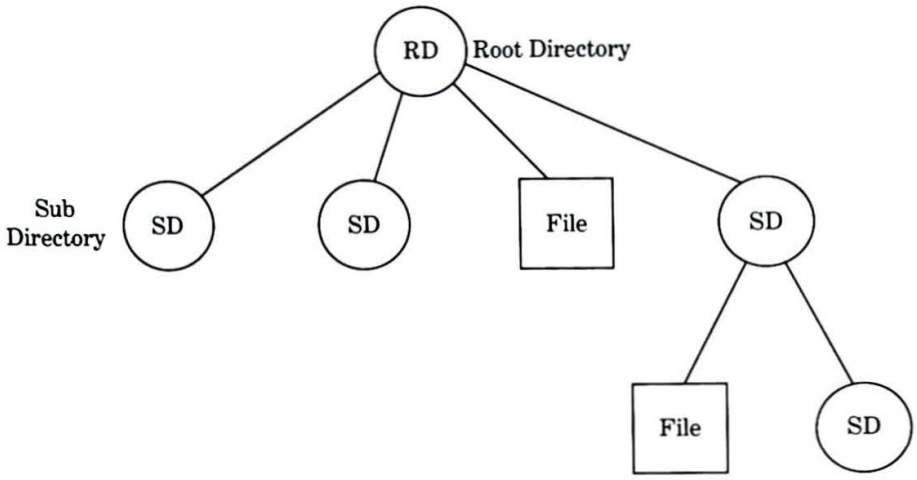



**FIGURE 3.1** Data structure.

presents the problem that a new job may not be loadable from disk, even though there is available memory space, because the particular required address space is occupied.

The efficiency problems of the batch monitor are addressed in software-managed memory systems by *partitioned allocations*, as shown in Figure 3.2(b). Given that real memory is large enough to hold a number of partitions, each partition is allocated its own address space in real memory. The monitor then manages these partitions, assigning space, adjusting addresses, and swapping partitions in and out. As jobs are completed, space in memory is vacated and the monitor attempts to load another partition. The problem with this system is primarily that memory becomes fragmented and is used inefficiently.

Figure 3.2(b) shows a 30-K job (Job 5) waiting to be allocated. Even after Jobs 1 and 4 are complete (the assumed completion sequence) Job 5 cannot be allocated, even though 35K of memory space are available. Only after Job 2 is completed does space in memory become available for allocation of Job 5. Fragmentation clearly results in the inefficient use of memory.

Early research into solutions for the problem of fragmentation led to the identification of two types of fragmentation in a virtual memory system [RAND68, RAND69].

*External fragmentation* is caused in real memory when the allocation system does not relocate used space to free up unused space for allocation to a new task. Thus, the real address space becomes fragmented into groups of unusable spaces. (This fragmentation is the situation described above.)

*Internal fragmentation* occurs when the average space required for

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000149

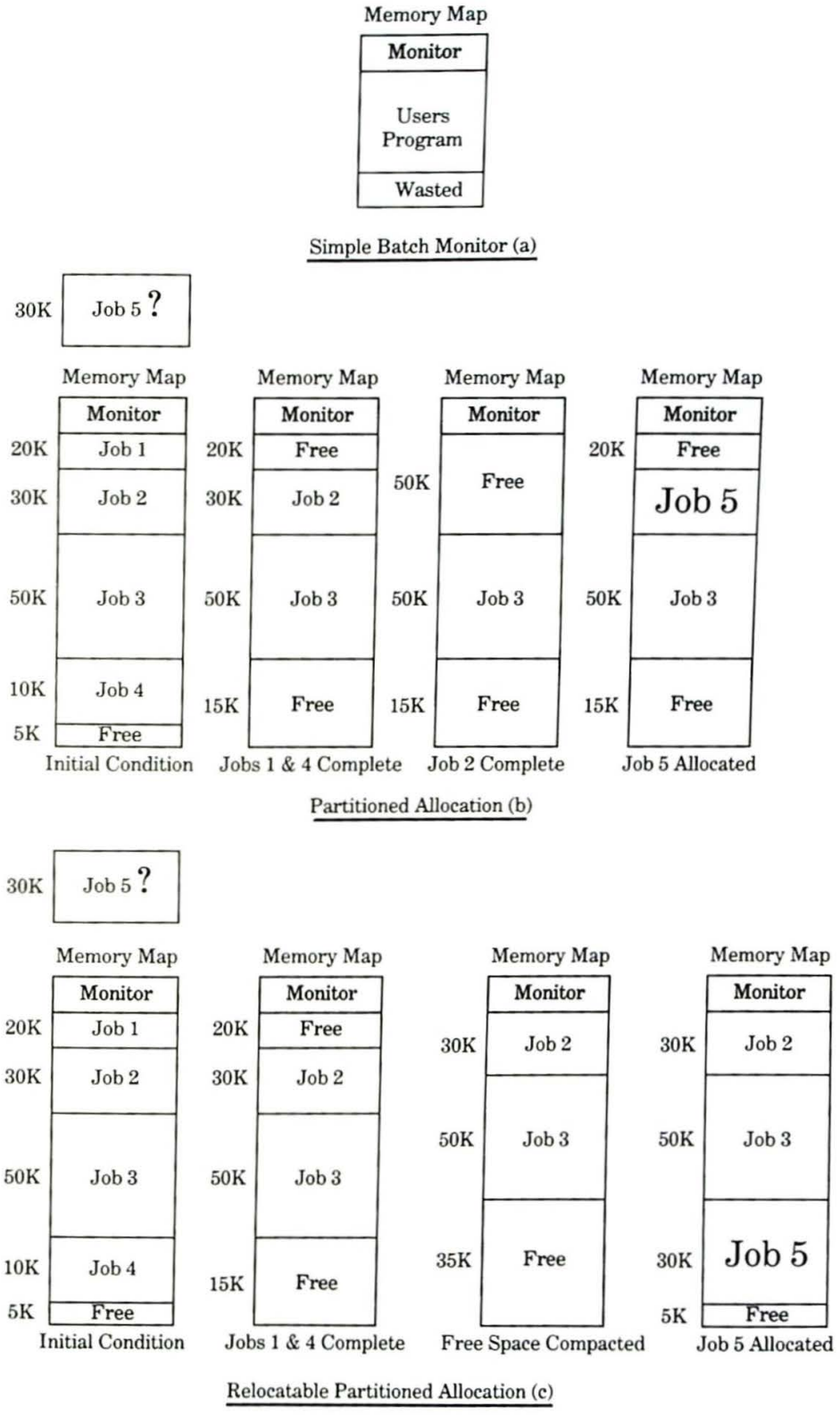



**FIGURE 3.2** Early software managed memory systems.



Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000150

an allocation is slightly larger than a page, which is a fixed-length allocation unit. Thus an extra page must be allocated, which wastes real memory space. Note that external and internal fragmentation were first applied in relation to the allocation of files on a disk. The meaning of these terms has been somewhat altered since they were applied to virtual memory systems. (This form of fragmentation is described later.)

External Fragmentation was solved by means of a *relocatable partitioned allocation*, which is illustrated in Figure 3.3(c). Relocatable partitioned allocation means that as jobs are terminated, the operating system compacts the remaining partitions so that free space is made available to waiting jobs. Note that after Jobs 1 and 4 are completed, the operating system moves Jobs 2 and 3 so that all free space is compacted into one address space in memory. This space is now large enough to allocate Job 5. Not only is memory used more efficiently by this process, Job 5 is available for execution earlier than it would be with the partitioned allocation system.

Relocation introduces an addressing problem, however. When partitions are moved, all addresses in the program must be changed. This problem is solved by the use of a *relocation register*, which points to the start of the partition that had been allocated under the assumption that the partition starts at address 0. All accesses to memory have the relocation register's contents added to the address. Relocation registers do not eliminate the need to actually move the partition, but they do minimize the addressing problems. Relocatable partitioned allocations go a long way to solve the external fragmentation problem but can introduce significant overhead because of the need to move the partitions around in memory.

*Paged allocation* extends relocatable partitioned allocations into a more generalized technique. Instead of a relocation register for each partition, paged allocation uses a relocation register for pages of approximately 4 Kbytes (size varies across systems). In some early systems the operating system managed these page relocation registers. Because the partitions are subdivided into pages, each having a relocation register, it is unnecessary to move pages as with relocatable partitions. A partition is broken into pages and placed into available page locations in the address space of real memory. The program can still be written under the assumption of a start address of 0 because the relocation registers change each address into the correct page location address. Paged allocation is the basic approach found in *demand paged virtual memory*, which is described in more detail later in this chapter.

*Segmented allocation* involves relocatable partitions of variable length and may be viewed as a variable-length page that supports the program's view of the data structure. Segmentation facilitates the sharing of programs and data.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000151

The problems of software support of the above-mentioned memory management schemes have led researchers to see if the processor itself could make these tasks transparent to the executing program, in other words, the development of *virtual memory*. The first description of a virtual memory applied to the Atlas computer [KILB62], a design motivated by the desire for a large linear address space that hid the memory management tasks from the programmer. Another early experimental paged system was implemented at Berkeley [LAMP66]. The IBM S/370 family, first delivered in 1970, extended the IBM S/360 family to paged virtual memory capability. Cocke [COCK59] described the "virtual memory" of Stretch; however, this scheme is in reality an early form of cache.

A "pure" segmented system (pure means that a complete segment is fetched at one time from disk) was first implemented in the late 1950s on the experimental Rice University computer. The Burroughs B5000, first delivered in 1965, was the first large-scale commercial implementation of virtual memory and used segmentation [LEVY84]. In England, ICL implemented a segmentation machine in the early 1960s [ILIF68].

For almost three decades, research activity has produced hundreds of research papers and reports. An authoritative early work on virtual memories is by Denning [DENN70], a paper that explains most of the tradeoffs in the design of a virtual memory system. [PARM72] is another review paper, describing work at IBM, with an annotated bibliography. This paper looks not only at virtual memory but virtual machine concepts as well. [DONO72] reflects the MIT Project MAC view of memory management and virtual memory.

## 3.1 Virtual Memory Concepts

The term *virtual memory* has been used to denote a computer with large linear address space that hides memory management tasks, discussed above, from the programmer. Virtual memory can be implemented with pure software or combinations of hardware and software. The following sections describe the use of hardware support to remove from the operating system some of the requirements that are necessary to manage the details of allocation. For the ensuing discussion, I define the following terms.

*Symbolic name* is the set of all names used by the programmer.

*Effective address* is the address generated by the processor and may, in some cases, be extended into a virtual address.

*Virtual address* is the machine equivalent of a symbolic name.

*Real address* (physical address) is the set of all addresses in the real memory.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000152

*Virtual memory system* translates a user's virtual addresses into real memory addresses.

*Allocation* is the process of reserving storage locations.

*Page* is an allocation of fixed-length aggregations of addressable units.

*Demand paging* is a virtual memory system that moves pages to real memory as needed rather than prior to need.

*Paging* is the process or moving pages back and forth between real and auxiliary memory.

*Segment* is an allocation of variable-length aggregations of addressable units.

*Paged segmentation* is a virtual memory system that uses segments composed of integer pages, not addressable units.

There are differences in vocabulary for virtual memory concepts used by different researchers and manufacturers. For example, the address presented to the virtual memory system is known as either *virtual address, logical address*, or *linear address*. Another example is found in the names for the address presented to real memory: *real address*, *main memory address*, or *physical address*.

The most common and serious error in computer design has been to design an address range that is too small [BELL76]. It seems that few processor designs can outlive this flaw without a significant extension to the address range. An extension of the address range has been accompanied by a transition to virtual memory, a move that is typified by the DEC transition from the PDP-11 to the VAX 11/780 and the IBM transition from S/360 to S/370. Dennis [DENN65] pointed out the rationale behind the need for a large virtual address space.

1. A computation should have the use of a name space sufficiently large so that all information it references may be assigned unique names and such that reallocation of information within its name space is never necessary.
2. Data objects of a computation should be expandable without requiring a reallocation of name space.
3. Information referenced in common by several computations should have the same name for all computations that reference it.
4. A protection mechanism should operate in name space to permit access to information by a computation only in an authorized manner.

These requirements established the direction of virtual memory development. Research into techniques for the manipulation of symbolically addressed files with software was the precursor for virtual memory. This research established most of the concepts now embodied in virtual memory. As noted in [DENN70] four research objectives were: (1) program modularity, (2) varying data structures, (3) protection, and (4)

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000153

sharing. The virtual memory systems discussed in this chapter deal with the way in which systems can be implemented in hardware and software to provide these functions.

When a program with symbolic names is compiled, two address translations are performed. First, the symbolic names are translated into virtual addresses that are used by the processor to address the virtual memory. Second, the operating system allocates the program and data to the disk system providing addresses such as drive, head, track, and record (discussed further in Section 3.5.9). The translations are

Compiler/Operating system translations.

Symbolic names-*translated to*-Virtual addresses,

Symbolic names-*translated to*-Disk addresses.

For a system with a cache, a program reference to memory may require three run time translations. The address translations, called Maps by [MATI77], are

Run Time translations.

Virtual addresses ⇒ MAP 1 ⇒ Cache addresses,

Virtual addresses ⇒ MAP 2 ⇒ Real memory addresses,

Virtual addresses ⇒ MAP 3 ⇒ Disk addresses.

MAP 2 and MAP 3 are the major subjects of this chapter. Translatio. to a cache address, MAP 1, is discussed in Chapter 4. Caches (discussed in Chapter 2) are used to improve the performance of the memory system by reducing the latency of the main memory as seen by the processor. Virtual memory, on the other hand, reduces the capacity limitations of small real memories by bringing the disk system into the address space of the processor. In early systems, the capacity gap is managed by programming with overlays. That is, the programmer divided the program and data into modules and scheduled their movement from the disk to memory. A virtual memory automatically manages this operation and presents a "one-level" memory to the user.

In addition to, and perhaps more important than, the one-level memory concept, virtual memory implementations have evolved to support useful data structures and programming methods. A multi-user system will have a number of active programs with requirements for sharing and protection that can be satisfied by virtual memory. Multiprogramming systems (*multitasking*) also require the same sharing and protection capabilities. In addition, programmers want to be able to develop and test individual programs that would not interact with other programs. These issues all stimulated the development of virtual memory concepts. There are three ways that virtual memory can be provided.

1. The processor can perform virtual memory functions completely in an interpretative mode. The speed penalty, as great as a factor of 100, may result when using interpretative virtual memory.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000154

2. The compiler/loader can perform the virtual memory functions by relocating all addresses prior to execution (discussed in Section 3.0).
3. There can be hardware support for some of the virtual memory functions that overcome many of the performance penalties of pure interpretative mode. Combined hardware/software virtual memory is the predominate technique and is described in the sections that follow.

The implementation concepts of virtual memory and caches are similar. The differences are in the selection of the design parameters and the use of software rather than the hardware implementations of caches. Because the speed ratio between the real memory and the disk is quite large—100,000 to 1 or greater, there is abundant time for the use of software-implemented policies when there is a miss in the main memory; see Figure 1.7. Another difference between virtual memory and a cache is the terminology. Virtual memory and caches were developed almost concurrently, but they are described by different researchers using different terminology.

The basic idea behind virtual memory is that main memory is a cache for the disk just as the cache, discussed in Chapter 2, gives the illusion of a fast main memory. Thus, the discussion of virtual memory will address the same two issues that are addressed with caches:

1. organization and structure of the memory hierarchy (discussed in Section 3.2),
2. set of rules to follow when main memory is accessed (discussed in Section 3.5).

As with caches, the translation of address spaces is directly related to the organization of the virtual memory system. The process of converting the processor effective address into the real memory address is performed in two steps, as shown in Figure 3.3.

The first step, sometimes called *mapping*, takes the processor effective address and extends it into the virtual address (discussed in Section 3.2.1). Some processors, however, use the term *segmentation* for this first step; examples are found with IBM RS/6000 and Intel x86. Some processors, by virtue of their long processor effective address, do not extend to a longer virtual address.

The second step, also shown in Figure 3.3, translates the virtual

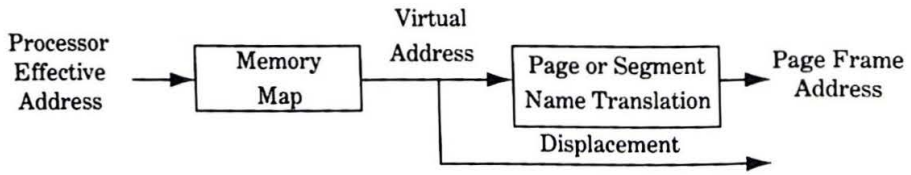



**FIGURE 3.3** Address conversions.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000155

page name into the *page frame address* (PFA) or segment name into a base register value to be combined with the untranslated displacement into the real memory address; a step is called *name* or *address translation*. Early systems, such as the IBM 7040, used the processor effective address as the memory real address. In these cases, the mappings are identities. Some contemporary systems have, as part of their virtual memory management hardware, an identity name translation path for accessing special system areas in memory such as name translation tables.

Virtual memory is similar to caches because the major functions to be provided are the same. In addition, they are early select because of the process of performing the name translation. These functions are

1. to detect the presence or absence of the referenced AU in real memory;
2. if the AU is present, to translate the virtual address into the real address and access the real memory;
3. if the AU is absent, to determine if a page or segment must be evicted from real memory as well as select the page or segment to be evicted;
4. to evict the page or segment and bring in the new page or segment;
5. to update all translation tables and reinitiate the access.

These functions are tightly interwoven into the design of what is now known as the *Memory Management Unit* (MMU). The discussion of virtual memory starts, in Section 3.2, by describing the methods that extend the effective address into the virtual address. The method by which the page or segment name is translated is discussed in Section 3.3. A major consideration in the design of a virtual memory is the reduction of the time to make the above-mentioned translations. The other virtual memory functions are discussed in Section 3.4.

## 3.2 Memory Mapping

Most early computers and microprocessors suffered from limited address space. Examples include the PDP-8 with 12 bits of address and the i8088 with 16 bits of address. Short addresses limit the amount of random access memory that can be attached, and a number of techniques have been employed to extend the address range of processors. A complete survey of these techniques can be found in [POPP77]. Invariably, these extension techniques produced a granularity in addressing that complicated the programming.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000156

### 3.2.1 *Creating Large Virtual Addresses*

This section discusses the problem of creating a virtual address for systems where the processor space or effective address is less than the desired virtual space. The page or segment name portion of the large virtual address is then translated into the real memory address. The inadequacy of a 32-bit virtual address has been recognized for three decades and extensions have been applied to 32-bit addresses. For example, the IBM S/370/168 (circa 1965) had a 3-bit extension of a user ID that extended the virtual address. The creation of large virtual addresses is more commonly viewed as creating a set of large segments that are of a fixed maximum length and can overlap. For this reason, the mapping function is frequently referred to as *segmentation*.

In general, extensions are found in a map that is indexed by a small number of bits in the effective address or indexed by implication. The steps used to extend the effective address are usually accomplished in one level of mapping.

### 3.2.2 *One-Level Map Address Extension*

The first use of maps to extend an address is found in short word-length minicomputers of the 1970s. For these machines, the virtual address is too short to address the desired real memory directly. Thus, a map is used to extend the virtual address [POPP77]. An example of this technique is found in the Z800 memory system, as shown in Figure 3.4.

**Z800**. This processor generated a 16-bit effective address. The 4 MSBs are used as an index into a 16-entry table containing control

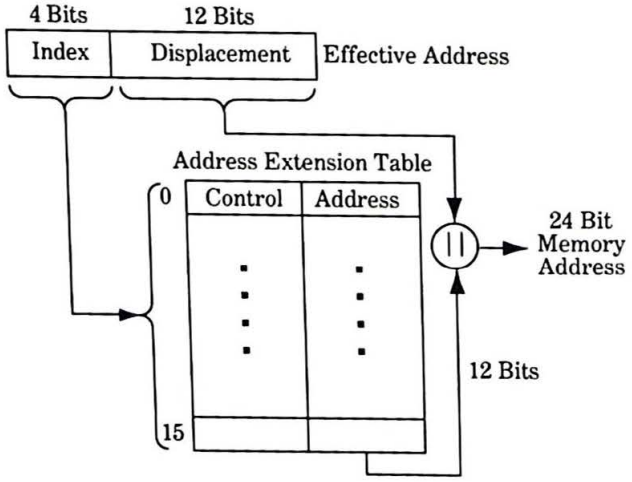



**FIGURE 3.4** Z800 address extension.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000157

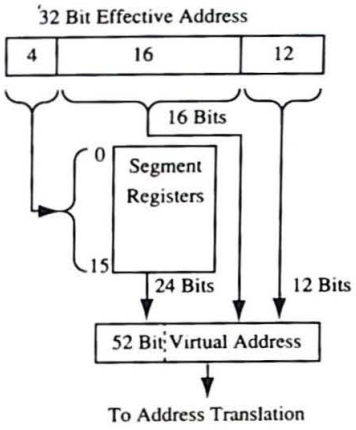



**FIGURE 3.5** RS/6000 MAP.

bits, for access control, and a 12-bit address extension. The address and displacement are concatenated to provide a 24-bit memory address.

**RS/6000**. This processor takes a similar approach to generating a long virtual address from its 32-bit effective address. The virtual memory system of this processor is directly related to that of the IBM 801 [CHAN88, OEHL90]. With this processor, the effective address is 32 bits and is mapped into a virtual address of 52 bits. Figure 3.5 shows the mapping of the processor effective address into the virtual address.

The four most significant bits of the effective address are an index used to select one of sixteen segment registers that store 24-bit segment identification numbers (SID) or addresses. These 24 bits are concatenated to the 16 virtual page indices, which creates a 40-bit virtual page name. These 40 bits are concatenated with the 12-bit displacement, creating the 52-bit virtual address. The address extension mapping provides 16 active segments of $2^{28}$ bytes, or 256 Mbytes, at any one time. The total number of segments in virtual memory space is $2^{24}$. During a memory reference, the 40-bit virtual page name is translated into a real address as discussed in Sections 3.3 and 3.4.

**HP Precision Architecture**. The HP Precision Architecture [MAHO86, FOTL87] generates a 32-bit effective address. The two MSBs index into a four-entry map that contains 0-, 16-, or 32-bit extensions (depending upon the processor model). These extension bits are concatenated with the remaining 30 bits of the effective address to produce a 30-, 46- or 62-bit virtual address. The page is 2 Kbytes; thus there are $2^{19}$, $2^{35}$, or $2^{51}$ virtual pages.

**Motorola MC68040**. This processor extends its 32-bit effective address by implication indexing one of two 23-bit registers and concatenating it with the effective address. There is one register for user and

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000158

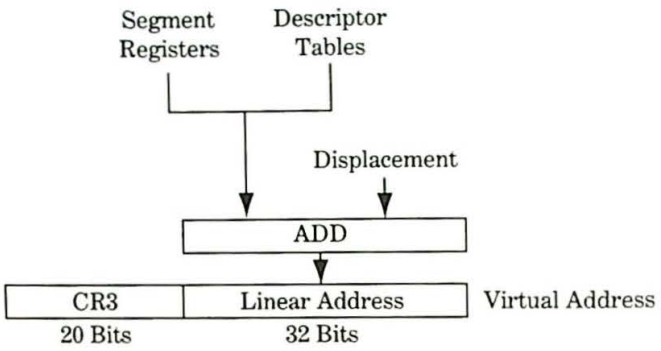



**FIGURE 3.6** i386, i486, and Pentium register based address extension.

one for supervisor mode. The result of this operation is a 55-bit virtual address. With a page of 8 Kbytes, there are $2^{42}$ virtual pages. The contents of these extension registers are called *root pointers* because they point to the root directory in the virtual memory space.

### 3.2.3 *Register Extensions*

The i386, i486, and Pentium extend their 32-bit segmented address (the generation of this address is described in Section 3.3.3) by 20 bits to form a 52-bit virtual address as shown in Figure 3.6. This extension is taken from CR3, one of four control registers (CR0–CR3) in the processor, that is loaded by the program using move instructions. The extension is called the *page director base register*, which is sometimes called the *root pointer* because it points to the root directory in the page tables.

### 3.2.4 *Summary*

Many processors extend their effective address as discussed above. Extension is used, in some cases, to extend the address space in conjunction with providing support for paged segmentation. Table 3.1 lists the number of extension bits and the number of virtual pages that can be addressed. In all cases, except for the IBM S/370/168, the effective address is extended from 32 bits.

The extension registers are loaded under program control and, in some designs, provide segmentlike virtual address on top of a paged system. In these cases, the management of the segments is a pure software function. Paged segmentation is discussed in Section 3.3.3.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000159

| Processor | Extension Bits | Page Size | No. Virtual Pages |
|---|---|---|---|
| RS/6000 | 20 | 4 Kbytes | $2^{40}$ |
| MC68030,40 | 23 | 8 Kbytes | $2^{42}$ |
| MC88200 | 20 | 4 Kbytes | $2^{40}$ |
| i386 and i486 | 20 | 4 Kbytes | $2^{40}$ |
| HP Precision | 0,16, or 32 | 2 Kbytes | $2^{19}$, $2^{35}$, or $2^{51}$ |
| MIPS R2000 | 6 | 4 Kbytes | $2^{26}$ |
| IBM S/370/168 | 3 | 2 Kbytes | $2^{15}$ |

**TABLE 3.1** Effective address extensions.

## 3.3 Virtual Memory Organizations

A basic function performed in a virtual memory system is the translation of virtual addresses to real addresses. A number of methods for performing the name translation function have been used in virtual memory systems, and a taxonomy of these methods is shown in Figure 3.7. The taxonomy shows the three major types of virtual memory—paged, segmented, and paged-segmented—and the translation methods used for each. The sections to follow discuss the translation.

Two design features are common to paged and paged-segmented virtual memory organizations: (1) congruence page mapping, and (2) early selection (see Figure 2.11). Within these two design features, the translation of a virtual address into the real memory address is performed by accessing *tables* that contain preloaded translation information, which is an operating system function. Segmented systems that

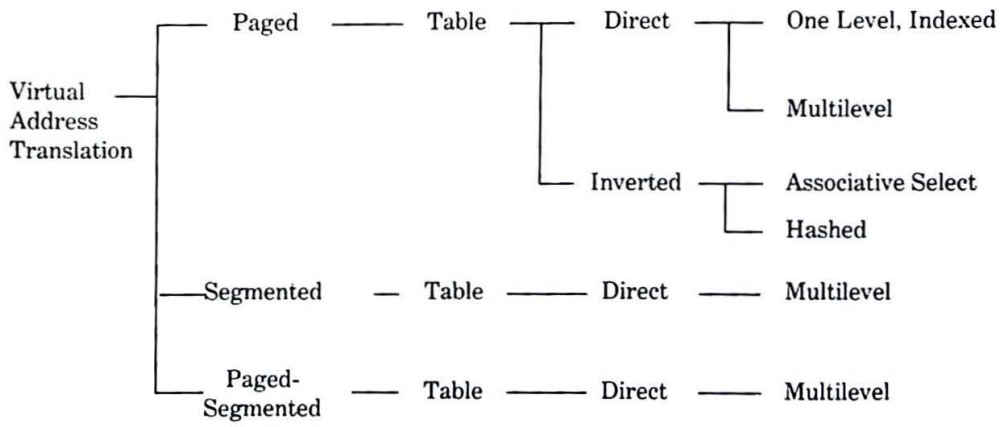



**FIGURE 3.7** Virtual memory and translation taxonomy.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000160

also use a translation table for name translation are early select but are not congruence mapped.

The use of early select techniques in virtual memory systems is based on the following consideration. The real memory, unlike a cache, is quite large. Thus, the number of tag bits needed for a late select real memory presents a significant cost problem as the implementation of the tag memory could best be accomplished in the memory technology itself. Early selection requires a number of memory cycles for name translation but, like most references, would be served by the cache; very few references require this slow translation process. Some of the terminology, expanded from earlier definitions, used with virtual memories include the following.

A *page* is a length allocation of bytes that is transferred to/from the real memory and the disk, similar to a block in the cache. Pages are congruence mapped. Consequently, pages do not overlap within the virtual memory address space.

A *segment* is a variable-length allocation or page. However, there is usually a limit to the size of a segment. The segment is addressed from a base register, and segments can overlap partially or fully. Segments are not congruence mapped and can overlap.

A page and segment can be defined by the Cartesian product of fixed- or variable-length and overlap or nonoverlap as

| | |
|---|---|
| Fixed length, nonoverlap | Page |
| Fixed length, overlap | Segment |
| Variable length, overlap | True segment |
| Variable length, nonoverlap | Not defined |

A *page frame* (*page slot*) is a page-sized space in real memory that can hold a page.

A *page fault* occurs when a requested AU is missing from the real memory; similar to a cache miss.

The *page fault ratio* is the ratio of unsuccessful accesses of real memory to total accesses of real memory.

A *demand paging system* is a hardware implementation of virtual memory.

The taxonomy presented in Figure 3.7 is based upon the structure of the translation mechanism. Moore [MOOR91] provides other taxonomies. One of his taxonomies is based upon the granularity of the data structure: word, page, segment, and page-segment; while a second taxonomy is based upon the allocation and management methods: manual static, manual dynamic, compiler, linker/loader, and automatic dynamic.

Another consideration in the organization of the name translation

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000161

system is the inclusion of control bits. These bits are required for maintaining the context of the translation system [MALI90] and are in addition to the pointer or address bits found in the translation table entries. The following sections discuss the control bits for each of the virtual memory organizations and describe the methods for performing name translation. Note that most systems also provide direct untranslated paths, or identity translations, used by the system software and the operating system.

The reader should keep in mind the changes in name translation techniques that have occurred over time. Early systems with small effective or virtual address spaces and small virtual memory used hardware-based translation tables that are fast and implementable with the available technology at reasonable cost. As the size of the address space and virtual memory increased, these tables were placed in memory at a significant cost in increased access time. To overcome this problem, small hardware-based translation systems called a translation lookaside buffer are used so that a large fraction of the translations are made in fast hardware. The translation lookaside buffer is identical to the early hardware-based translation systems. Today's virtual memory systems are an amalgam of the early hardware-based translation combined with memory resident translation tables.

### 3.3.1 *Page Allocation Systems*

For page allocation systems, the following translation of the virtu address into the real memory address is taken from [DENN76]:

$$\text{virtual address} \Rightarrow (n, d) \Rightarrow (f(n), d) \Rightarrow f(b) \ \emptyset \ d = \text{real address}$$

where $n$ is the page name; $d$ is the displacement within the page; $\emptyset$ denotes the concatenation; $(n, d)$ = [virtual address/s] or virtual address mod $s$, [ ] denotes the quotient of dividing the virtual address range by $s$, which is a trivial step when $s$ is an even power of two; and $s$ is the size of the page.

This translation process describes congruence mapping. That is, the virtual address is de-allocated into the pair (page name, displacement). The page name is translated with function $f$, and the result is concatenated with $d$ to give the real memory address. Note that the displacement is NOT translated. Because virtual memory systems are early select, the mapping is associative and any page can be placed in any page frame of real memory. Each of these methods of table-based page

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000162

name translation, as shown in the taxonomy of Figure 3.7, are discussed in the following paragraphs.

A major virtue of a paged system is that upon a page fault, it is only necessary to find space in real memory for a fixed-size page as the address translation process is congruence mapped and associative. This policy, however, does lead to some waste of real memory space because a segment of the data structure is not always an exact number of pages. This waste is known as internal fragmentation, which was previously defined.

## Table Translation

Translation of the page name of a virtual address into a real address is accomplished by the use of tables, called *page tables* or *maps*, that contain the translation information. There are three types of table-based translators.

*Direct page table.* A direct translation table has one entry for every page in virtual memory and is accessed by one or more indexing steps (discussed in Section 3.3.1).

*Inverted page table.* An inverted translation table has an entry for every page that is resident in real memory and is accessed by some form of associative or hashed index search [CHAN88].

*Cache-like page table.* A number of structures are used that take advantage of the temporal locality of addresses and use smaller tables than inverted translation tables. These tables that hold the translation information for a subset of the pages in real memory are generally called *translation lookaside buffers* and are discussed in Section 3.4.

Direct table translation is sometimes known as *forward-mapped translation*, and inverted translation is sometimes known as *reverse-mapped translation* [HUCK93]. Note that the word *direct* has a different meaning when used to describe cache organizations. There are variations in translation systems based on page, block, instruction, and data name translations. These variations will be discussed in the following sections.

## Direct Page Table Translation

A direct page translation table has *one table entry for each virtual page.* The page table is direct mapped and addressed by an index of the virtual page address that is the name of the virtual page [BAER80, WILK68]. With a large virtual address space, the direct page table can get quite large because of the large number of virtual pages. The variations discussed later address this issue.

The direct page tables discussed in this section may be viewed as a data structure, held in memory and manipulated by the processor's

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000163

program. However, in the interest of decreasing the time required to perform a page name translation, hardware tables and other hardware assists can be provided. For example, the traverse of the tables is called table-walking and, in some processors, is implemented in microcode.

*One-Level Direct Page Tables*

A one-level indexed direct page table is also known as a *linear map*. For processors with a small virtual address space, these tables have been implemented completely in hardware. A direct page table is illustrated in Figure 3.8. This example assumes a virtual memory of eight pages and a real memory of four page frames. The page table is indexed by the page name from the virtual address and has eight entries, each of which consists of a present bit plus the frame address. A page can be placed in any page frame; thus the translation is associative.

The state of the system shown in Figure 3.8 is: page 4 is resident in page frame 0, page 7 is in page frame 1, page 3 is in page frame 2, and page 5 is in page frame 3 of the real memory. If a reference is made to, say, page 7, the present bit (= 1) indicates that the page is resident and the page frame address is selected, concatenated with the displacement from the virtual address, and indexes into the real memory. This method of generating the real address is recognized as early select.

A page name that indexes the table and finds the present bit a zero causes a page fault. With a page fault, a real memory page frame must be selected to receive the new page based upon criteria discussed in

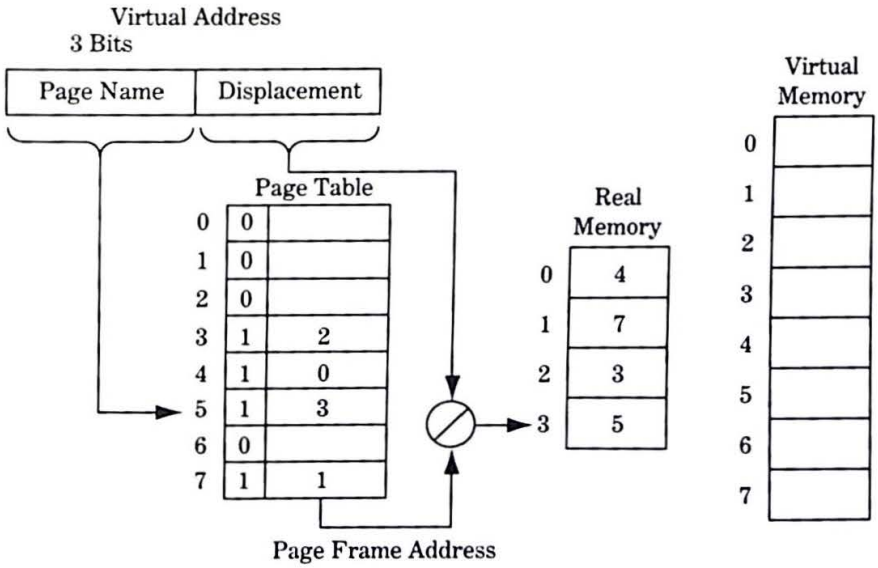



**FIGURE 3.8** One-level indexed direct page table.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000164

Section 3.5.2. If there is a vacant page frame, the requested page can be loaded into that frame and the present and address fields are updated. If there is no vacant page frame, a page frame must be selected based on some use criteria. If the page in the selected page frame is dirty, it must be written out to the disk before the new page is read in. After the page is loaded, the page frame address and the present bit are adjusted to indicate the presence of this new page in the real memory. No known implementations of a direct page table use a dirty bit to indicate a modified value in a page frame. The assumption is made that all page frames are dirty and must be evicted if selected. Page frames holding only nonmodifiable code would not need to be evicted, and a dirty bit could be of some value for these situations.

One of the first time-sharing virtual memory machines used direct translation [MEND66]. This machine, the Xerox Sigma 7, was developed by Scientific Data Systems based upon research performed at U.C. Berkeley. The processor effective address is not extended because it is relatively long (17 bits) for its day. The Sigma 7 had 256 virtual pages of 512 32-bit words, as shown in Figure 3.9. Real memory can have up to 256 page frames installed.

As Figure 3.9 illustrates, an 8-bit page name indexes into one of 256 page table locations containing a present bit and the 8-bit translated page frame address. Each page table entry also has protection bits, not shown, for memory access and write protection.

An advantage of one-level direct translation of a virtual address is its simplicity and a relatively short latency if the page table is in hard-

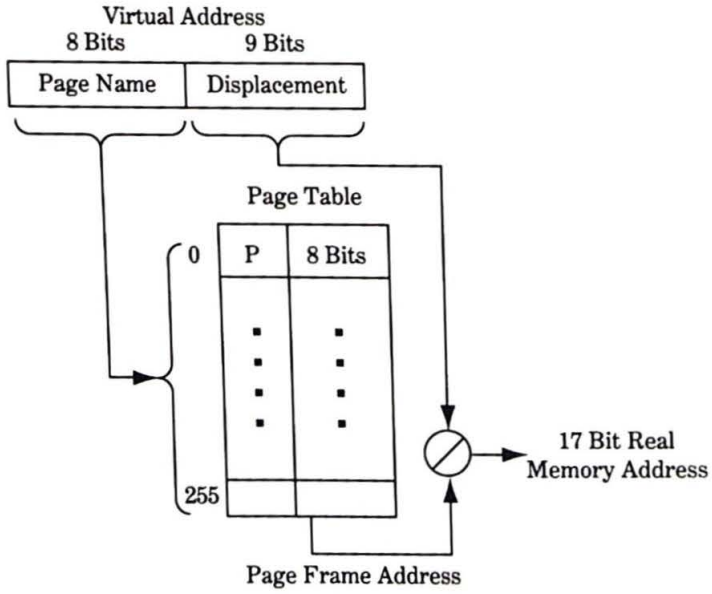



**FIGURE 3.9** Sigma 7 page name translation.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000165

ware registers. On the other hand, as the number of pages in the virtual memory gets very large, the size of the table increases to the point that there may be more bits in the table than in real memory. For example, assume a virtual address of 32 bits (address to the byte), a page size of 1 Kbytes, and real memory of 1 Mbytes. The number of entries in the page table is

$$\text{number of entries} = 2^{32}/2^{10} = 2^{22}.$$

The number of page frame address bits in each entry of the page table is $\text{Log}_2\ 1024 = 10$ bits. For this example, the page table has more than five times the number of bits as the real memory (5.5 Mbytes versus 1 Mbytes). As the virtual address size increases, the size problem with a one-level page table increases as well.

It is possible to store a one-level direct page table in virtual memory space that is itself paged. The table described above would occupy approximately 0.1% of the virtual address space. In other words, the real memory acts as a cache for the page table, greatly reducing the real memory space required to hold the page table. An example of this technique is found in the VAX 11/780 page table organization. Three base registers hold base addresses for three page tables that are indexed by the sum of a value in a base register and the virtual page name [STRE78]. Not only can portions of the page tables be paged, but portions of the page tables can be in the cache. In other words, the table structure is treated as any other data structure insofar as allocation to the memory hierarchy is concerned. This one-level page table is supported by a translation lookaside buffer (discussed in Section 3.4).

*Multilevel Direct Table Translation*

Multilevel direct tables, sometimes known as *hierarchical tables*, have a consistent relationship to the structure of the program and data as illustrated in Figure 3.10. The multilevel tables are arranged in a tree structure that contains two or more levels of partial direct page tables [ARDE66]. The first level is generally associated with the root node of processes that are resident in real memory, while the second and third or fourth levels provide the name translation for the individual process segments. With this structure, some degree of protection between processes is provided. The advantages of the multilevel page table technique over the single-level indexed direct technique have been discussed in [FURH87] and include

1. more sophisticated protection,
2. accommodation of a larger address space,
3. page and segment sharing between processes.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000166

Protection is discussed in Section 3.5.7. The second issue of the accommodation of a larger address space is addressed in the following paragraphs and is a major benefit of the structure of multilevel translation tables.

Recall that direct page tables have one entry per virtual page and can become quite large with large virtual addresses. Because pages have locality, it is possible to restructure a one-level page table into a hierarchy (multiple levels) and page the active portions of the page table into real memory. In other words, the active portions of the page table are "cached" in the main memory so that accesses to the disk are not required to perform page fault detection and page name translation. While caching does not save table bits—in fact, it requires more bits—only the first-level root and the active levels, including the leaf levels, need to be in real memory at any time. Note that the leaf-level tables are usually referred to as the *page tables*. As will be shown, multilevel page tables permit the dedication of a relatively small portion of real memory to the "caching" of the active portions of the page tables.

A one-level indexed page table can be restructured into a multilevel page table of any number of levels, as illustrated in Figure 3.10. The structures of one-, two-, and three-level systems are illustrated. The page name is $N$ bits (the displacement field of the virtual address is ignored for simplicity), and the number of page frames in real memory is $2^P$. The one-level page table has $2^N$ words—each having $p$ bits of page frame address that index into the real memory. These $p$ bits are concatenated with the displacement, as shown in Figure 3.8. Control and present bits are in addition to the $P$ address bits and are not shown or counted.

For this model development, a two-level page table divides the page name into two equal parts. The root directory table, the first level, has $2^{N/2}$ entries of $N/2$ bits each. The first-level table is indexed by the most significant half of the page name bits. There are now $2^{N/2}$ second-level page tables, each with $P$ bits. The second-level table is selected by the address bits in the first-level table and indexed by the least significant half bits of the page name.

Note that there are more bits in the two-level system than in the one-level system. The leaf tables, in sum, have the same number of bits as the one-level system. The root-level bits are added to the leaf-level bits to give the total number of bits in the set of tables.

A three-level system is formed by dividing the one-level page table into $2^{2N/3}$ leaf tables, each have $2^{N/3}$ entries with $P$ bits. A second level consists of $2^{N/3}$ tables of $2^{N/3}$ entries having $2^{N/3}$ bits. The root level has $2^{N/3}$ entries of $N/2$ bits each. The three-level system also has more bits than the one level due to the addition of the root- and first-level to the third-level bits.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000167

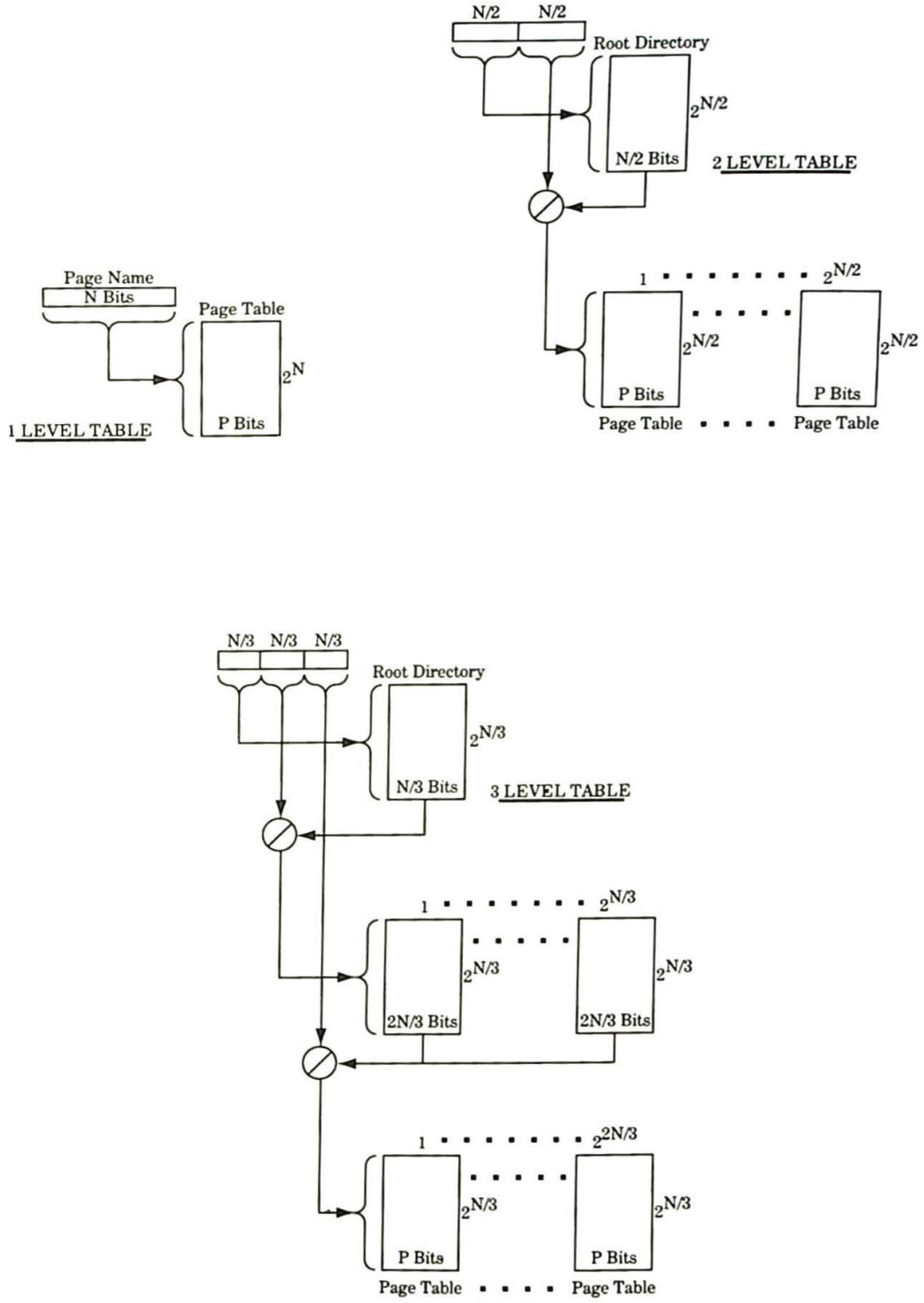



**FIGURE 3.10** Multilevel page tables.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000168

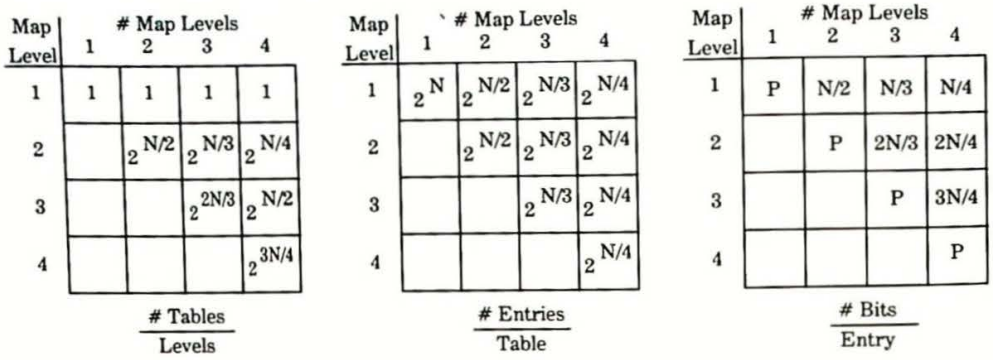



**FIGURE 3.11** Multilevel table parameters.

Figure 3.11 shows the number of tables per level, the number of entries per table, and the number of bits per entry for one-, two-, three-, and four-level multilevel page tables. These parameters are based on page name lengths with factors of 2, 3, and 4. With real systems it is not always possible to factor the page name and the number of bits shown in Figure 3.11 will not apply exactly.

As shown in Figure 3.11, a one-level page table has $2^N$ entries of $p$ bits. A three-level page table, for example, has $1 + 2^{N/3} + 2^{2N/3}$ tables, and each table has $2^{N/3}$ entries. The first-level page table has $N/3$ bits; the second level has $2^{N/3}$ bits; and the third level, leaf, page tables have $p$ bits. The total number of bits in an $l$-level page table is given by the formula [HABA93]

$$\text{total bits} = 2^{N/l}\left[\left(\sum_{x=0}^{l-2} 2^{xN/l}(x+1)\frac{N}{l}\right) + 2^{(l-1)N/l}P\right]$$

where $l$ is the number of levels.

Matick [MATI77] gives the following formula for the number of levels of a multilevel page table that is set by the page size, the number of virtual pages, and the number of AUs required for an entry in the page tables.

$$\text{number of levels} = \left\lceil \frac{\text{no. of V.PageNameAddressBits}}{\text{no. of PageAddressBits} - \lg_2 \dfrac{\text{no. of AUs}}{\text{Entry}}} \right\rceil$$

As the number of AUs per entry increases, fewer entries can be held in a given size page and the number of levels of the page table must increase. While each leaf table is small, the total storage of all the leaf

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000169

tables is the same as for a one-level direct translation system. The number of entries is $2^N$, as shown in Figures 3.10 and 3.11, a fact to be noted when multilevel page table systems are discussed.

What has been accomplished by subdividing a one-level page table into multiple levels? Each of the leaf tables has become relatively small. Subdividing is similar to block size selection in cache design. Smaller leaf tables will have a shorter transfer time from disk to the main memory but will also have a higher "miss ratio." Because portions of the page table are "cached" in real memory, there can be misses on the access of the page table itself. A page table miss is detected by a valid bit in the page table entry at each level except the leaf level.

The benefit of multilevel table translation is that large virtual address spaces can have translation tables that occupy a relatively small portion of real memory because only the active page table entries are resident. As a result, the miss ratio on the page tables, after the load transient, can be quite small. With portions of the multilevel page tables located in real memory, special provisions must be taken to prevent undue performance degradation when accessing these tables. Each level of the page table takes at least one memory reference to access the table itself, thereby increasing the latency of the page name translation process. Reasonably fast techniques, called *table walking*, for searching these tables are needed and are discussed in Section 3.5.5.

*Multilevel Page Table Control Bits*

The control bits required in a multilevel page table differ depending upon the level in the hierarchy. A one-level system or the leaf tables of a multilevel table has control bits required to manage the access to the page in memory, while the non-leaf tables have bits to manage the translation tables. Examples of the control bits used are

Leaf Table.

*Modified* or *dirty bit* indicates that one AU in the page has been written into and that the page must be written back into the virtual memory space if the page frame is to be vacated.

*Referenced bit* indicates that an AU has been referenced. This bit is required to support replacement algorithms.

*Protection bits* are required to ensure that the access method is correct—as is the case, for example, of Read Only, Read/Write, and Execute Only. Also, the access can be verified with protection bits for protected accesses such as User, System, or Operating System. These issues are discussed in Section 3.5.7.

Leaf and Nonleaf tables.

*Valid bit* or *present bit* indicates that the table entry is valid and/or that the referenced page is in real memory.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000170

Valid or present bits are used differently in different levels of multilevel page tables. Some systems, notably the i386, i486, and the Pentium, view the valid bits uniformly. That is, an invalid entry at any level signifies that the page is not in real memory. When an invalid bit is encountered in the translation process, a page fault interrupt is generated. Other systems, such as the MC68040, have a nonuniform treatment. The valid or present bits of the nonleaf tables indicate whether or not the reference to the next level translation table is valid. The bit in the leaf table indicates whether or not the referenced page is resident in real memory.

*Example Systems*

**Intel i386 and i486**. An example of a two-level multilevel table is taken from the i386 and i486 [ELAY85] and is shown in Figure 3.12. Note that the 32-bit processor address is known by Intel as the *linear address*. The linear address is expanded to 52 bits by concatenating the 20 MSBs of processor register CR3. The virtual address is divided into three fields: two fields from the virtual page name and the displacement. The 30 bits formed from CR3 and the directory field index into the first-level page table, called the *page directory*. The second level produces the page frame address for the real memory access.

Note that the leaf nodes, page tables, have 1K pages of 1K entries. This page table design is clearly not a complete direct translation table system. If it were complete, with $2^{40}$ virtual pages, four levels of translation would be required and the total number of leaf entries would be $2^{40}$, not the implemented $2^{20}$ Also, because there can be only $2^{10}$ unique

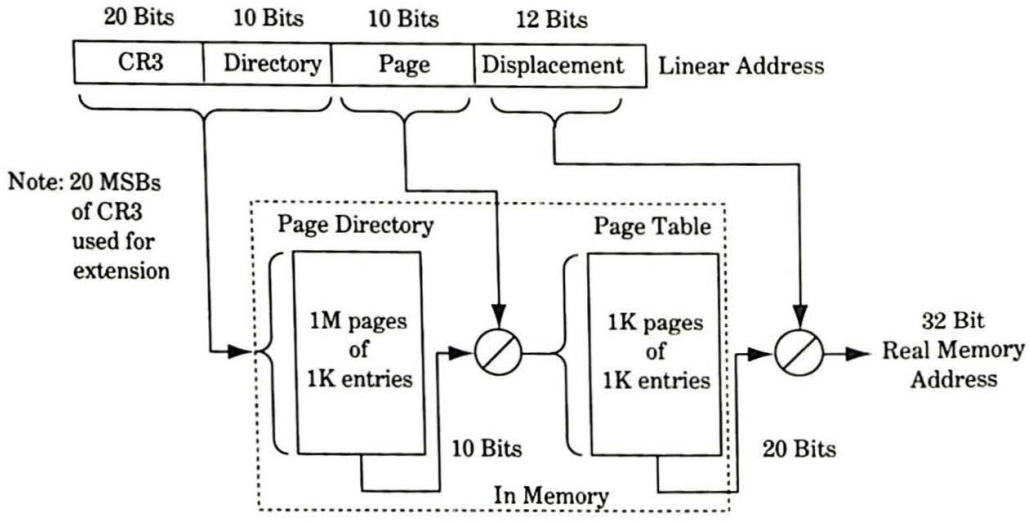



**FIGURE 3.12** Two-level page table (i386, i486).

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000171

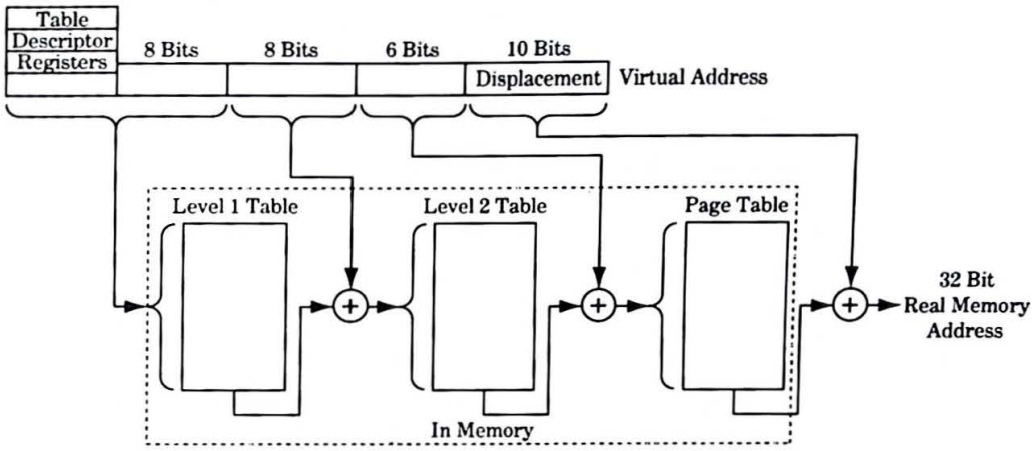



**FIGURE 3.13** Three-level page table (Z80,000).

instances of page frame addresses in the page directory of $2^{30}$ entries, address aliasing is possible. The design compromise made between a full or partial implementation is based, it is believed, on the desire to reduce the translation time via the page tables.

**Zilog Z80,000**. An example of a three-level page table system is found in the obsolete Z80,000, as shown in Figure 3.13 [FURH87]. The 32-bit virtual address is divided into a 10-bit displacement and a 22-bit page name address and is extended with bits from one of the four table descriptor registers that are addressed by two bits in the processor status word. The page name address is divided nonsymmetrically with an 8-bit index into the first level, an 8-bit index into the second level, and a 6-bit index into the third level.

Note that the Z80,000 page name translation system adds the index from the virtual address to the base from a table. The consequence of this is that entries in the page tables can overlap, providing for sharing but a more complicated software management system. Because of the use of addition, this system could be classified as paged-segmented; however, it is referred to in the literature as a pure paged system. The design details of this mapping system are not available, making it impossible to determine the size of the tables and other design details.

**MC68851**. The Motorola 68851 MMU is a stand-alone device intended for use with the MC68020 microprocessor [MILE 90]. This device is unique in that the structure is highly programmable or configurable by the user. The following structural parameters are available:

| | |
|---|---|
| Page size | 256 bytes to 32 Kbytes, |
| Number of levels | 1 to 5. |

An unusual feature found in the M68851 is the capability of storing

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000172

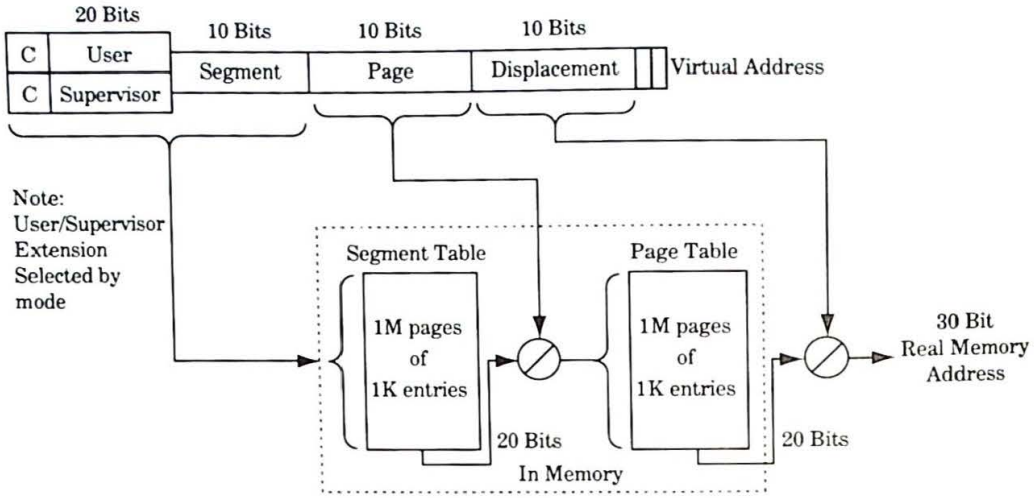



**FIGURE 3.14** MC88200 table structure.

a pointer in the leaf tables that point not to the page in real memory but to another leaf table. This feature supports sharing of real pages but can significantly increase the number of references to memory that are required before a page fault is detected. I believe that there is a time-out feature to stop endless looping around indirect addresses.

**MC88200**. It is interesting to note that Motorola eliminated the programmability of the MC68851 in the MMU of the MC88200 [MOTO90a]. The table structure of the MC88200 is shown in Figure 3.14. This device supports a fixed page size of 4 Kbytes. The 32-bit effective address is extended by 20 bits using two registers found in the MMU that are loaded by the processor. Depending upon the mode of operation, the user of supervisor extension is concatenated to the effective address, providing a 52-bit virtual address. There are two levels of translation. The first level, called the segment table, has $2^{30}$ entries of 20 bits that are concatenated with the 10-bit page field that address the page table. Motorola calls their system a segmented system; segmentation is provided by software that determines the values to be used in the extension registers. The balance of the system is a hardware-paged system.

As with the i386, i486 page name translation systems, the MC88200 is a partial implementation of a direct map. There are $2^{30}$ entries in the page table rather than $2^{40}$ as required for a full implementation. The number of entries is greater than that of the i386, i486, thus requiring fewer allocations of translation information to the page table.

### *Control Bits*

The use of control bits in a multilevel page table system is illustrated with the MC88200, which has an extensive control bit repertoire to

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000173

provide flexible support of differing operating systems. The first level of control bits is contained in the two *area descriptors* that are selected by mode to extend the processor effective address, as shown in Figure 3.14. These bits, noted as *C* in the figure, consist of

| | |
|---|---|
| WT | Write Through: controls the write policy of the cache; |
| G | Global: controls the scope of snoopy coherency measures (Chapter 5); |
| CI | Cache Inhibit: inhibits caching of local instructions; |
| TE | Translation Enable: if not set, no page name translation. |

The first-level table, known as the *segment table*, has entries called *segment descriptors*, with the following control bits:

| | |
|---|---|
| WT | Write Through; |
| G | Global; |
| CI | Cache Inhibit; |
| SP | Supervisor Protection: controls translation of supervisor; |
| WP | Write Protect: controls write protection; |
| V | Valid: if not valid, translation fault (table not present). |

The second level, known as the *page table*, has entries called *page descriptors*, with the following control bits:

| | |
|---|---|
| WT | Write Through; |
| G | Global; |
| CI | Cache Inhibit; |
| SP | Supervisor Protection; |
| WP | Write Protect; |
| V | Valid (Page is valid); |
| M | Modified: indicates that the page has been modified; |
| U | Used: indicates that a page has been accessed. |

Note that some of the levels have control bits for the same function. An example is the WT, Write Through bit. These bits are ORed together, and if any one is set the resulting action is taken based on this test of the control bits.

### Inverted Page Table Translation

Direct multilevel table translation has served well for processors with virtual addresses limited to 32 bits. With virtual address extensions, partial page table implementations have been required. In the early 1980s designers at IBM contemplated very large virtual addresses and

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000174

determined that a different translation method would be required. For a system with a 1-Kbyte page and a 52-bit virtual address, the leaf nodes would require approximately $6 \times 2^{42}$ bytes. Systems that followed this study are the IBM System 38 with a virtual address of 48 bits [HOUD80, HOUD81] and the IBM RS/6000 with a 52-bit virtual address [OEHL90]. A direct page table for a large virtual address would not only be very large but also sparsely populated and inefficient. The inverted page translation systems described in this section are both small and densely populated.

To reduce the size of the translation table, an inverted page table that contains *only the virtual pages that are currently resident in real memory* can be used [CHAN88] to translate long virtual addresses. An inverted page table can provide a substantial reduction in the size of the page tables and reduce the number of memory accesses required for a page name translation. The page name can be translated into the real address by either an associative search of the translation table, an $n$-wise set associative search, or by hashing into a linked list. Current design practice does not use $n$-wise set associative searching, thus associative search and hashing are discussed in the following paragraphs.

All known contemporary implementations of inverted page table virtual memory have been designed by either IBM, Hewlett-Packard, or the IBM/Motorola/Apple group (Sumerset). These systems will be described in the following sections.

### Inverted Associative Page Table Search

An inverted *associative page table* (APT) has one entry for each of the pages in real memory. This page table is accessed by an associative search from the page name in the virtual address. As with all known virtual memory systems, an APT system is early select and congruent mapped.

The Atlas [KILB62, MATI77] is an example of an inverted APT. This memory system has a page size of 512 words, a virtual address of 2048 pages, and a real memory of 32 pages (16K words). A block diagram of the map is shown in Figure 3.15. The dotted arrow with *A* at its head signifies an associative search. The page table contains the page frame address and the page name of the pages in real memory, called *tags*. A page name is the key for an associative search of the tags. Note the one-to-one correspondence in the number of entries in the page table and the number of page frames in real memory.

A hit on the tags produces a 5-bit page frame address that is concatenated with the 9-bit displacement from the virtual address. The resulting 14-bit address is presented to the real memory. If the addressed page is not in real memory and therefore not in the page table, a miss occurs and the page is fetched from a drum memory in virtual address space.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000175

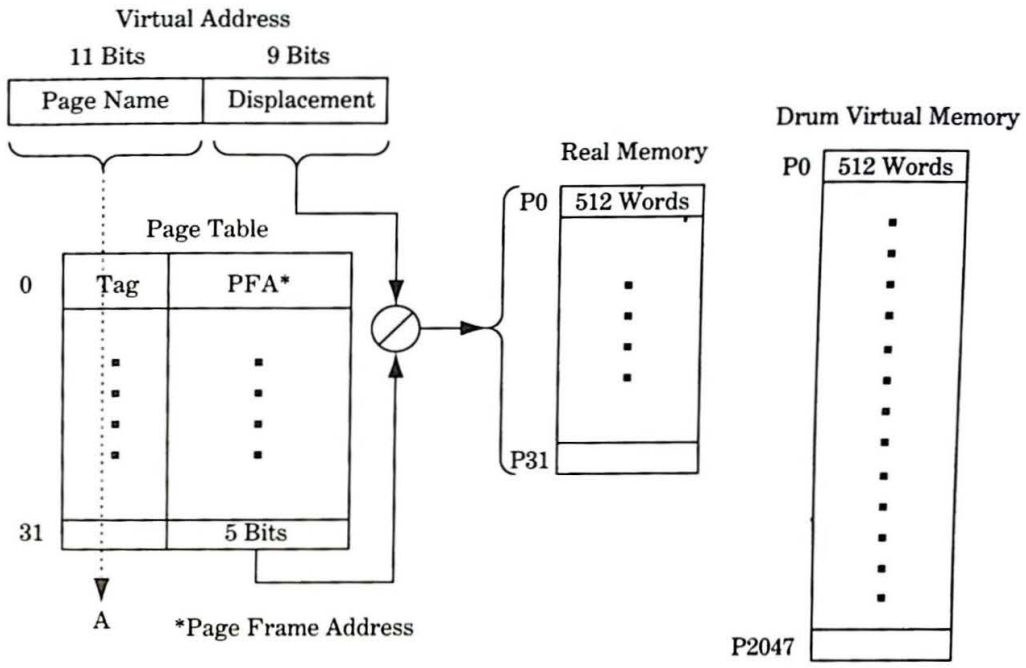



**FIGURE 3.15** Atlas page name translation.

The drum address is found by sequentially searching a table holding the 11-bit virtual page number and the corresponding drum address (a process described in Section 3.5.9).

The designers of Atlas considered using a direct table method with the table in real memory. However, this form of translation requires that at least one real memory reference must be processed to see if there is a page fault and to generate the real memory address. This design would make every reference to real memory cost (at a minimum) two memory cycles. The APT method provides a faster associative search of a hardware table, adding only a small increment of time to the real memory access latency. Because it is faster, the associative method is superior and is used on the Atlas even though there was a significant circuit cost for the associative translation table.

For very large virtual addresses and large real memory, the size of an inverted APT is too large and costly for consideration. The small virtual address of the Atlas will never be used again. However, the APT is used in systems in the form of translation lookaside buffers (discussed in Section 3.4).

*APT Control Bits*

Systems using APT control the page name translation process by an associative search mechanism and require no control bits. Recall that there is one table entry for each real page frame. The entries in the

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000176

inverted page table contain the page base address in real memory and the page name. If there is a hit on the search, the translation is complete. If there is a miss, then there is a page fault.

Protection bits should be required with the APT as with the other page name translation systems. However, as protection was not a requirement for APT systems that were implemented, such as the Atlas, there are no examples to cite of protection bits. Protection bits are used in TLBs (described in Section 3.4), which are similar to APTs. For example, valid bits are required to indicate the presence of valid translation information in the page table.

### *Hashed Inverted Page Table Search*

The inverted translation scheme of the Atlas is costly to implement if the size of the real memory and the virtual address become very large. This is now the case with many virtual memory systems. Hashing is the translation technique of choice today for IBM systems with large virtual addresses. The index into an inverted page table is found by hashing the page name (the displacement bits are not hashed). The hash number is then used as an index into the page table, which provides the page frame address. This technique is properly known as *key transformation* [PRIC71]. The key transformation technique reduces the large number of virtual page addresses into one of a number of noncongruent classes in which the real page addresses are linked together.

Hashed access into large address spaces or data structures was developed in the 1960s for access to symbol tables and the like [JOHN61, MORR68]. These techniques were then applied to the accessing of large files and have been, since the late 1970s, applied to virtual memory page name translation. A good hash transformation is one that "spreads the calculated address (sometimes known as hash addresses) uniformly across the available addresses" [MORR68]. The design of hashing functions is beyond the scope of this book, however, a tutorial on hashing can be found in [LEWI88].

It is possible to have a one level translation table that is accessed by hashing the virtual page name as illustrated in Figure 3.16. This method for translating virtual addresses into real memory addresses divides the virtual address into the page name field and the displacement field. The page name field is hashed for an index that accesses the page frame table. The table contains entries for all of the pages that are resident in real memory—an inverted translation system. The page frame table entry carries the page name *tag* and the page frame address of the page in real memory. When the page table is accessed the tag is compared to the page name and, if the comparison is true, the page frame address is gated out and concatenated with the displacement to

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000177

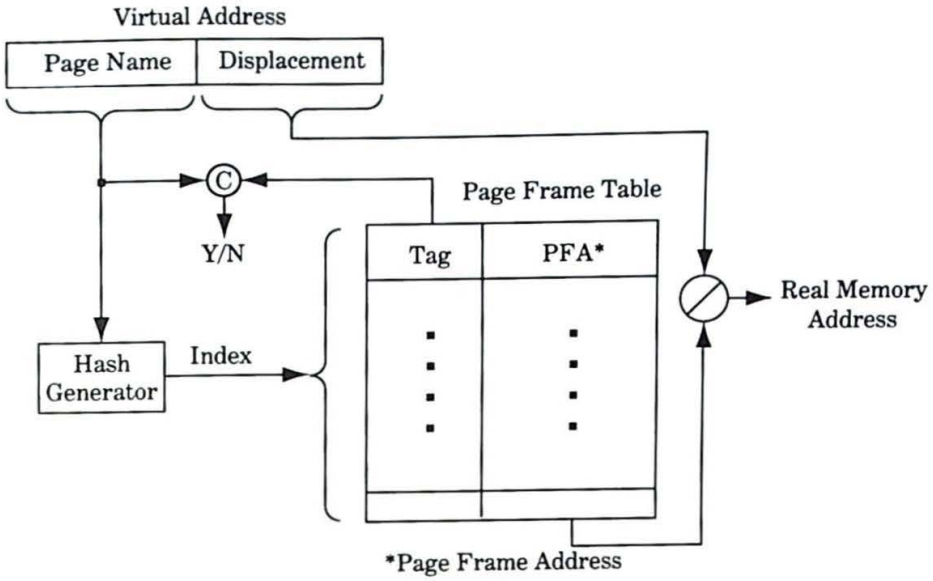



**FIGURE 3.16** Hashed page name translation.

form the real memory address. If the comparison is false, the addressed page is not in real memory and a page fault is signaled.

A problem with this simple hashed translation is that two or more virtual addresses can hash into the same value, creating a collision into the same page frame table entry. These collisions increase the page name translation fault rate, significantly increasing the average page name translation time. A number of techniques for managing collisions and reducing their performance impact have been developed and implemented in real systems [MORR68]; a list of these techniques is given in Table 3.2 with examples noted. These examples will be described in the following paragraphs.

*One-Level Large Page Frame Table*

**HP Precision Architecture**. A system that uses a one-level page frame is the Hewlett-Packard Precision Architecture as shown in Figure 3.17

| Technique | Example |
|---|---|
| One-level large page frame table | HP Precision Architecture |
| One-level linked list | None |
| Two-level linked list | IBM S/38–RS/6000 |
| One-level sequentially indexed list | PowerPC 601 |
| One-level disjoint collision table | Proposed [HUCK93] |
| $n$-way set associative | IBM S/360/168 TLB |

**TABLE 3.2** Collision handling techniques.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000178

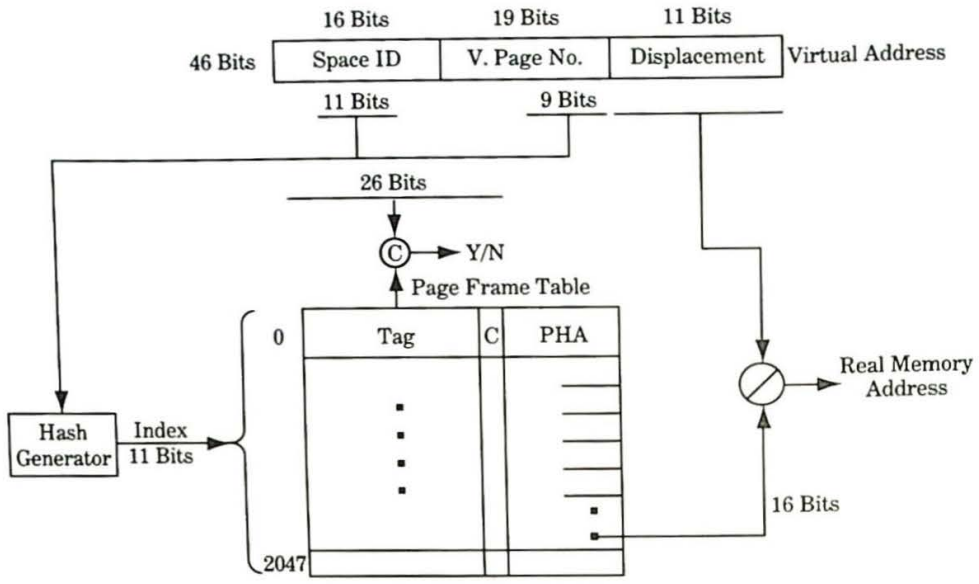



**FIGURE 3.17** HP PA page name translation.

[MAHO86, LEE89]. This system depends on reducing the probability of a collision by implementing a large page frame table. Because the translation time through this system is quite fast, a translation lookaside buffer is not used. However, Hewlett-Packard describes this system as a translation lookaside buffer without another table structure for handling TLB misses.

The virtual address is composed of an 11-bit displacement, a virtual page number (either 19, 35, or 51 bits), and a 16-bit space ID. Nine bits of the virtual page number and 11 bits of the space ID are hashed into an 11-bit index. This index references a one-level, direct page frame table, called a translation lookaside buffer by HP. An entry contains a 26-bit tag that is compared to 26 bits in the virtual page name. The table produces a 16-bit page frame address that is concatenated with the 11-bit page displacement. A collision (no match on the tag but with a true valid bit) requires a full virtual software page name translation.

### *One-Level Linked List Page Frame Table*

The approach to collision management discussed in this section is commonly used today because it provides a good balance between speed and cost. A one level system that mitigates the problem of hashing into the same page table entry by using a linked list is shown in Figure 3.18. Each entry of the *page frame table* (the name used in the IBM RS/6000) contains three fields: a page name tag, a page frame address, and a link field (*chain*). When a page is loaded into a page frame in real memory,

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000179

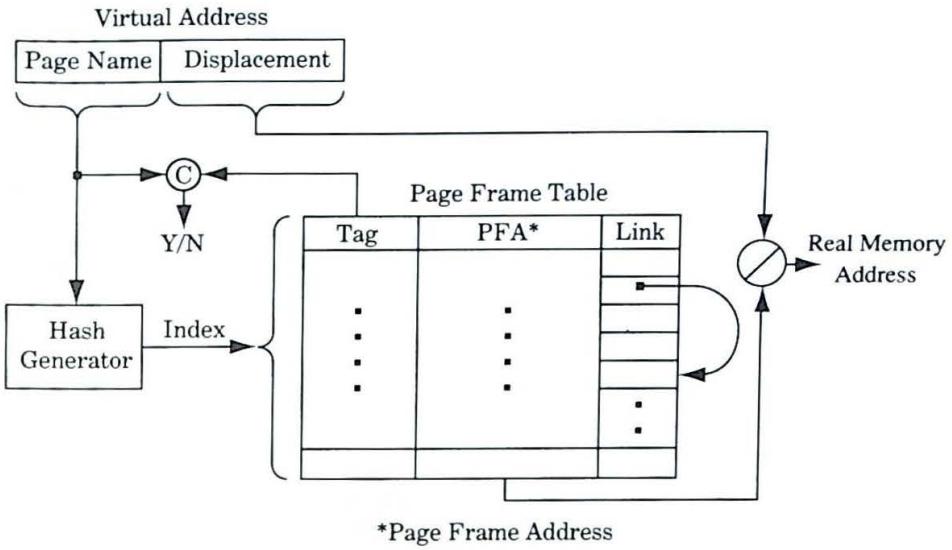



FIGURE 3.18 One-level hash accessed inverted page table.

its page name and the page frame address are placed into the page frame table.

A virtual memory page name translation is performed by hashing the page name to an index into the page frame table. The page name field of the addressed entry is compared to the page name of the virtual address. If they are the same, the correct page frame address is concatenated with the displacement to give the real memory address. If the page name is not the same, a hash collision has occurred and the link pointer accesses another entry, and so on, until there is a match on the tags or a terminator symbol is encountered in the last entry. If the last entry has been accessed without a match on the page names, the referenced page is not in real memory and a page fault results.

A one-level hash accessed translation suffers from a significant problem. For some update cases, the entries in the page frame table must be moved to make room for the allocation of a new entry in the page frame table. Moving entries in the page frame table consumes time and reduces the overall performance of the system. Allocation in a one-level system is illustrated in Figure 3.19, which shows the states of the page frame table before and after allocation. Before allocation we see three linked lists of page translation information. Note that the page names *A* and *B* both hash into "1", the page name *D* hashes into "8", and the page names *P*, *Q*, and *R* hash into "17". In this example, the page frame table entries have been allocated into consecutive locations following the hash address and are linked via the link fields. A valid bit is needed in each entry to identify the page frame table entries that are allocated or vacant.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000180

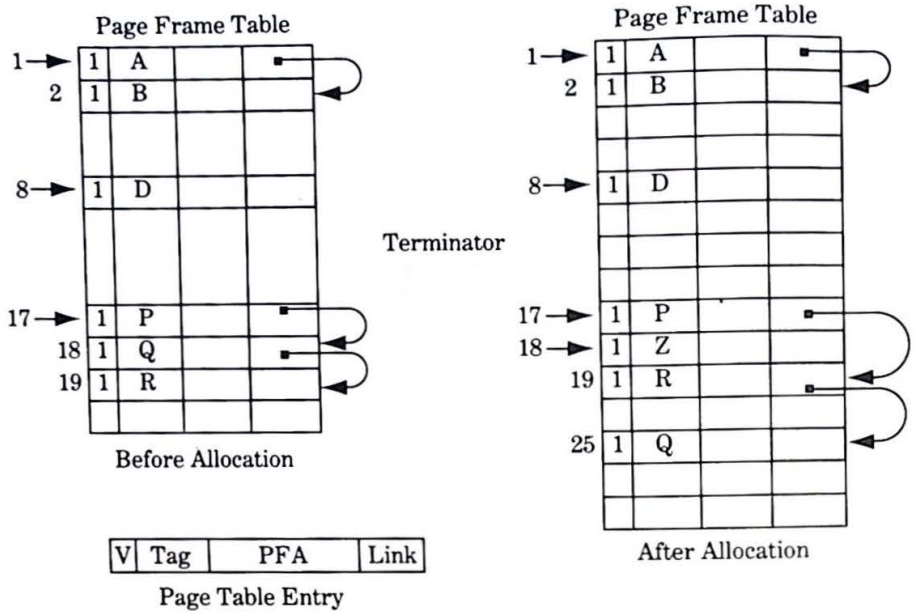



**FIGURE 3.19** Allocation with a one-level hash accessed inverted page table.

Consider what happens if a new page is allocated and its page name "Z" hashes to the value "18". The entry "18" in the page frame table is already occupied by Q and must be moved in order to vacate location "18". This can be accomplished by moving the Q entry into location "25" and adjusting the link pointers. A vacant entry is found by scanning the PFT until a "0" is found, signifying a vacant table entry that can be allocated.

If there is a second allocation of a page with a page name that hashes into "18", its translation information is allocated to a vacant page frame table entry and appended to the linked list that starts with "18". Another situation exists for a new allocation hashed to an unused entry in the page frame table (the valid bit is 0). For this case, the tag and address information are allocated and the link field is loaded with the terminator symbol. This is a simple case, and no entry movement is required.

### *Two-Level Linked List Page Frame Table*

A solution to the problem of page frame table movement is to introduce another level in the translation process that, in effect, provides an indirect pointer to the translation information in the page frame table. With a level of indirection, allocating new entries to the page frame table does not require that other entries be moved. This first-level table, named a *scatter index table* in the early literature, is known as a *hash index table* in the IBM System 38, a *hash table* in the IBM 801, and a

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000181

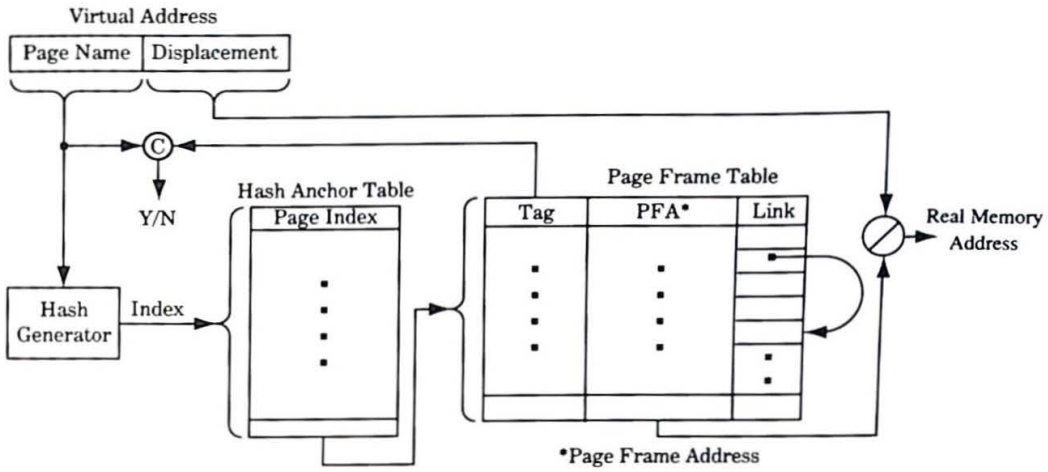



**FIGURE 3.20** Two-level hash accessed inverted page table.

*hash anchor table* with the IBM RT PC and RS/6000, which is the term used in this text.

A two-level system is shown in Figure 3.20 [JOHN61, MORR68]. With this system, the page name is hashed for an index into the hash anchor table, which contains an index into the page frame table. The page frame table contains the page name (that is compared to the virtual address page name), the page frame address, and a link. The page frame address is concatenated with the displacement to form the real memory address.

The allocation example in Figure 3.18 used for the one-level system is illustrated in Figure 3.21 with the before and after cases shown. Here, the hash anchor table contains a pointer into the page frame table. For this example, the linked list *A*, *B* is located starting at location 100 of the page frame table, *D* is located at 200, and the linked list *P*, *Q*, *R* starts at 300. A memory reference to pages *P*, *Q*, or *R*, for example, hashes into address "17" and the pointer 300 is found that indexes to location 300 of the page frame table. The proper page frame address is found by comparing the tags to the virtual page name and traversing the linked list if necessary.

When page "*Z*" is allocated, location "18" in the hash anchor table is vacant and the page frame table entry is placed in a vacant location, 250 for example, of the page frame table. No movement of page frame table entries is required to effect the allocation of the new page. If a page is allocated with a page name that hashes into an already used hash anchor table address, the allocation can be to any vacant location in the page frame table. It is only necessary to append the entry to the linked list. The more successful the hashing function is in providing a uniform

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000182

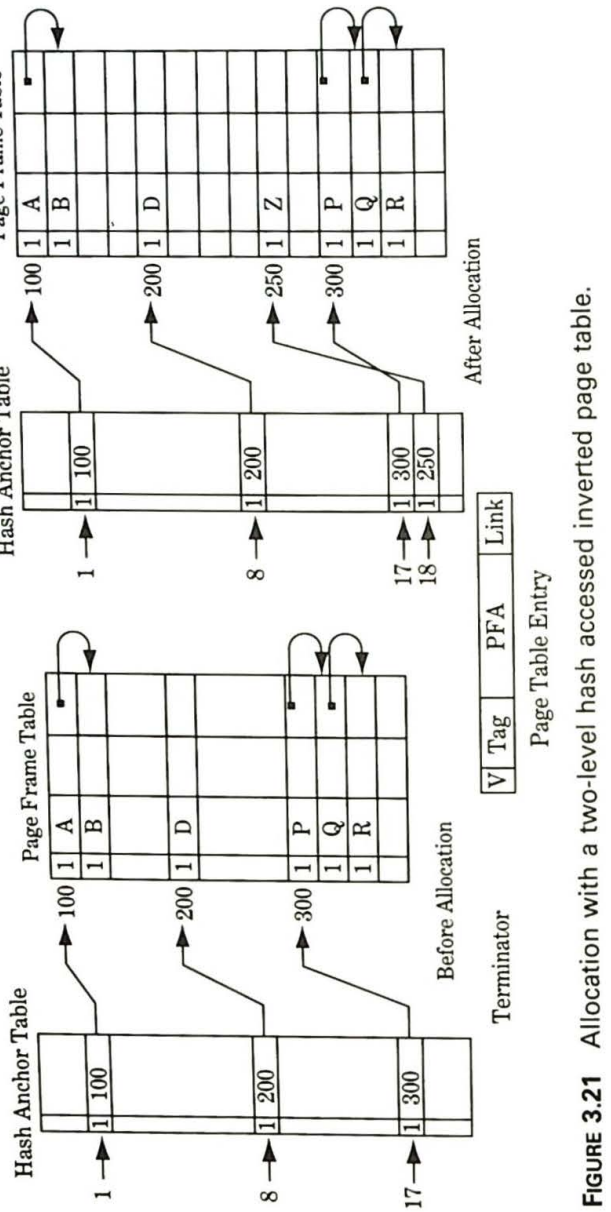



**FIGURE 3.21** Allocation with a two-level hash accessed inverted page table.



Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000183

distribution of transformed keys, the more balanced will be the length of the linked lists.

How many entries should there be in the page frame table and the hash anchor table? First consider the page frame table. Because an inverted page translation system has one entry for each page frame of real memory, the minimum number of entries is the number of real page frames. A large page frame table will have more vacant positions, reducing the length of the linked lists and the time required to find a page frame address.

The size of the hash anchor table is determined by the desired performance. A large hash anchor table has a larger hashed address, reducing the frequency of collisions in the page frame table. Johnson [JOHN61] determined that the average number of probes (reference attempts) into the page frame table is a function of the size of the page frame table and the hash anchor table:

$$P \approx 1 + \frac{\text{no. of entries in page frame table}}{2 \times \text{no. of entries in hash anchor table}}$$

where $P$ is the average number of probes into the page frame table translate a page name.

For example, if the hash anchor table has the same number of entries as the page frame table, 1.5 probes are required, on average, into the page frame table plus the access of the hash anchor table to translate a page name. If the hash anchor table has only 10% of the entries of the page frame table, six probes are required on average. An explanation of this model is found in the following consideration. If there is only one entry in the hash anchor table, every page name hashes into the same entry and all of the entries of the page frame table are linked together in one linked list. In this situation, the average number of probes approaches half the number of entries in the page frame table. This behavior is as expected from the time required to search a linked list sequentially. On the other hand, if the hash access table is very large, there are no collisions and each entry in the page frame table is unique (not linked). In this case only one probe is required.

Four IBM systems (S/38, 801, PC RT, and RS/6000) use a two-level hash access technique and are discussed in the following paragraphs.

**IBM S/38.** The virtual memory of the S/38 resulted from research at IBM in the late 1960s and early 1970s on the so-called Future System, which was abandoned due to lack of compatibility with S/370 [LEVY84]. The S/38 virtualized a segmentation system on top of the hardware-paged system [HOUD79, HOUD81]. The S/38 has a 48-bit virtual

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000184

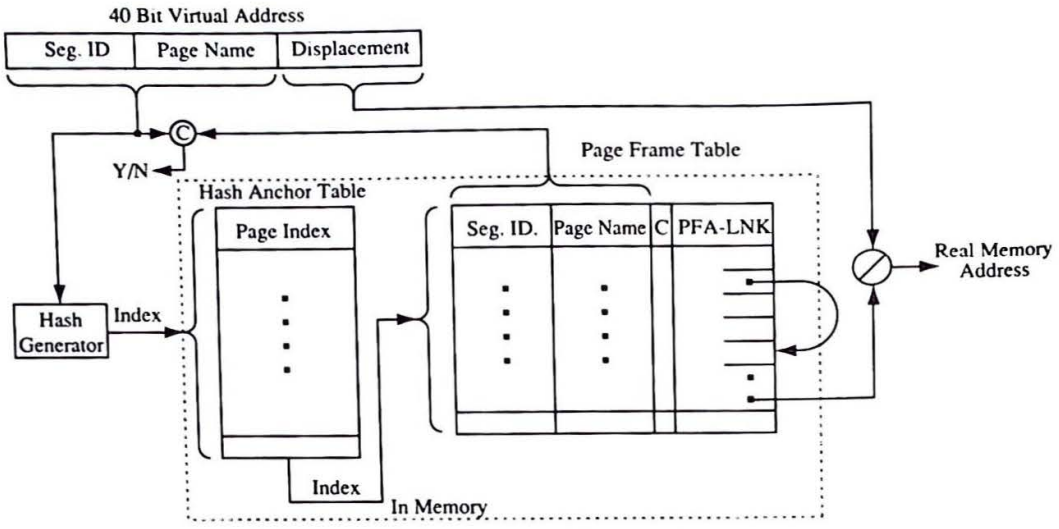



**FIGURE 3.22** IBM 801 page name translation.

address, with a 512-byte page. The 39-bit virtual page name is hashed by logical XORing three fields (two of which are bit reversed) in the virtual page name in order to generate a 7-bit index into the hash anchor table. A page frame table index that indexes into the 128-entry page frame table is produced, and the entries in the page frame table are linked together with pointers. Approximately 2.5 memory accesses are required to translate a page name in the page frame table; the total translation time is 3.5 accesses.

**IBM 801**. The inverted page table system of the S/38 evolved into the virtual memory system of the experimental IBM 801; its page name translation tables are shown in Figure 3.22 [CHAN88]. The IBM 801 generates a 32-bit effective address that is expanded to a 40-bit virtual address using a map that is similar to that adopted by the RS/6000 shown in Figure 3.5. Page name translation is via a hash anchor table and a page frame table stored in real main memory. Unfortunately the published literature does not give the sizes of the three fields of the virtual address.

The IBM 801 design combined the page frame address and link into one field of the page frame table. This field is noted in Figure 3.22 as the PFA-LNK field and is large enough to address any page frame in real memory (at the page level) or to address the next entry in the page frame table [CHAN88]. The operation of this combined field is described below.

A page name extended with a segment ID is hashed for an index into the hash anchor table, which then provides an index into the page frame table. If there is not a match on the page name, the link field

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000185

indexes into another entry. If there is a match, the link field is the page frame address in real memory and is concatenated with the displacement field from the virtual address. In other words, the link field is multiplied by 2 to the power of the displacement bits (left shifted the number of bits of the displacement). When there is not a match, the link field has its MSBs set to zero and the resulting value becomes the index into the page frame table. This process assumes that the page tables are allocated in the low address page frames of real memory.

Both the hash anchor table and the page frame table are stored in real memory. Therefore, a translation requires a minimum of two memory accesses to translate a page name. The size of the hash anchor table and page frame table are set when the system is generated depending upon the amount of installed real memory in the system. The hash anchor table can be set to be either equal to or twice the size of the page frame table.

**IBM PC RT**. The IBM PC RT also stores both the hash anchor table and the page frame table in memory [SIMP86, HEST86], as shown in Figure 3.23. The 32-bit effective address is expanded to a 40-bit virtual address by means of a 16-entry segment register. The page size can be set at either 2 Kbytes or 4 Kbytes, leaving either 28 or 29 bits that are hashed by an XOR operation for an index into the hash anchor table.

Experience with the IBM RT PC [CHAN88] shows that 2.5 storage accesses, or 1.5 probes to the page frame table, are required per page name translation. From this result we would assume that the hash anchor table has the same number of entries as the page frame table—a conclusion based on the Johnson [JOHN61] model. As with the IBM 801, setting the size of these tables is done when the system is generated.

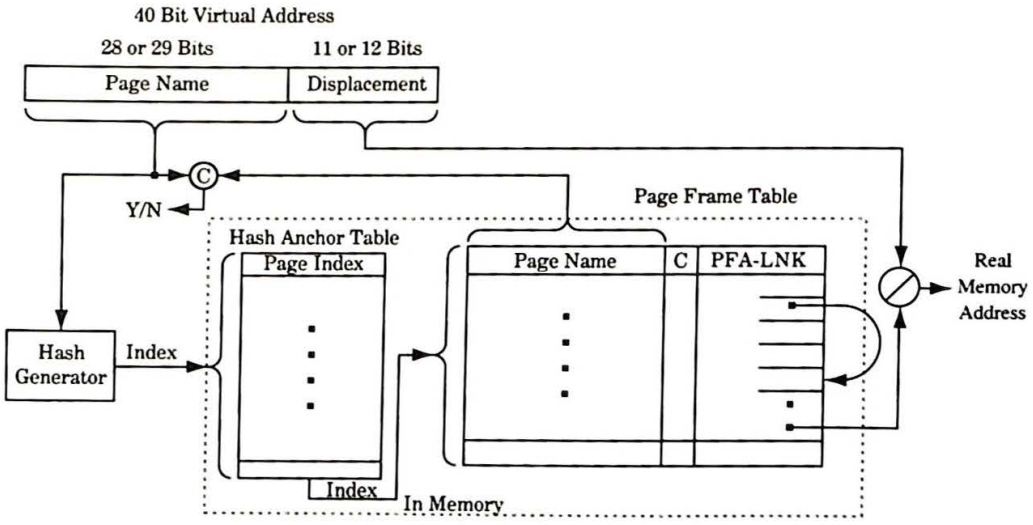



FIGURE 3.23 IBM PC RT page name translation.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000186

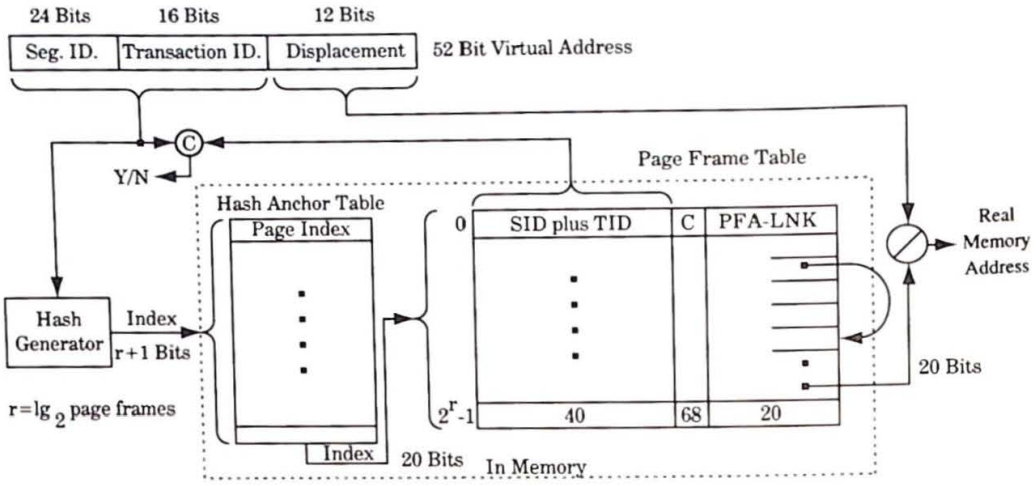



**FIGURE 3.24** IBM RS/6000 page name translation.

**IBM RS/6000**. The IBM RS/6000 continues the approach started with the IBM Future System development [IBM90]. Because the RS/6000 is a recently introduced computer, its virtual memory system is described in more detail than the other IBM systems described above. The RS/6000 can be viewed as a paged-segmented system with the segments defined by the address map of Figure 3.5; the balance of the system is pure paged. A block diagram of the RS/6000 page name translation is shown in Figure 3.24.

The 52-bit virtual address is divided into three fields: a 24-bit segment ID, a 16-bit transaction page ID, and a 12-bit byte displacement [IBM90]. The hash generator XORs the 24-bit SID and the 16-bit TID extended with zeros to 24 bits. The low-order $r + 1$ bits of the result index the hash anchor table.

The size of the hash anchor table is set to be twice the number of real pages installed on the system, while the number of entries in the page frame table is set equal to the number of pages in installed real memory, which can reach a maximum of $2^{20}$ pages. These table size selections are made by the operating system.

Each entry in the page frame table consists of 16 bytes (4 words). A 20-bit link field (PFA-LNK) links to the next entry if the comparison fails. If the comparison succeeds, the link value of the entry point to the succeeding comparison is the page frame address and is concatenated with the 12-bit displacement, providing a 32-bit real memory address. In other words, on a hit the link field is multiplied by $2^{12}$, while on a miss the link is used as the address into the page frame table in the low addresses of real memory.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000187

| | | S/38 | 801 | PC-RT | RS/6000 |
|---|---|---|---|---|---|
| Names | Level 1 | Hash Index Table | Hash Table | Hash Anchor Table | Hash Anchor Table |
| | Level 2 | Page Directory Table | Page Table | Inverted Page Table | Page Frame Table |
| Sizes | Level 1 | 128 | | r | $2^{r+1}$ up to $2^{21}$ |
| | Level 2 | 64 | | r | $2^r$ up to $2^{20}$ |
| No. of Real Memory Page Frames | | | | | up to $2^{20}$ |
| Page Size | | 512 bytes | | 2 K or 4 Kbytes | 4 Kbytes |
| Virtual Address | | 48 bits | | 40 bits | 52 bits |
| Max Real Memory Size | | | | $2^{22}$ bytes | $2^{32}$ bytes |

Note. $r = \lg_2$ (Number of page frames in real memory).

**TABLE 3.3** IBM hash accessed inverted page tables.

A linked list presents the danger of an infinite translation loop du to a programming error. That is, the links can create a circular list and, if there is no hit on tag comparisons, the translation can run forever. To prevent this from happening, a maximum of 127 probes are permitted before a search is declared a failure. The number of probes for the other systems described above is not known.

The IBM systems have evolved over a period of time. In order to illustrate this evolution, Table 3.3 contains a summary of the relevant parameters of the IBM systems discussed above.

As noted previously, the hash anchor tables and page frame tables are too large to be stored in fast hardware registers. Consequently, it is necessary to enhance the performance of these systems by using a translation lookaside buffer [HOUD81, CHAN88, OEHL90]. After a page miss, the requested page is loaded into the real memory. In addition, the TLB, the hash anchor table, and page frame tables are updated. A subsequent reference to this page will find the page name translation information in the TLB. However, later references may find that an address cannot be translated in the TLB and recourse to the translation tables is required. After this translation, the TLB is updated. TLBs are described in Section 3.4 and memory allocation is discussed in Section 3.5.2.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000188

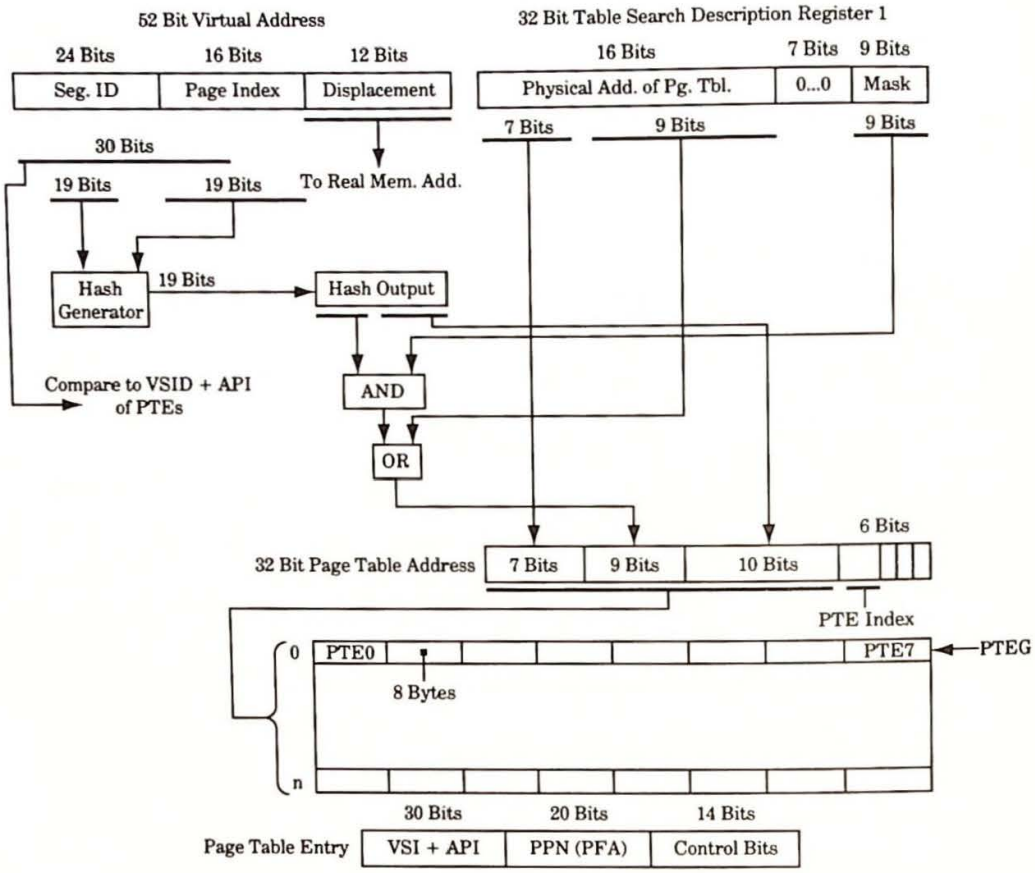



**FIGURE 3.25** PowerPC 601 page name translation.

*One-Level Sequential Indexed Page Frame Table*

**PowerPC 601**. The PowerPC 601 uses a hash-indexed inverted page name translation that resolves collisions by sequentially searching the page table entries. A simplified description of this system is given and illustrated in Figure 3.25; the terminology of [MOTO93] is used.

The virtual address is formed by a map that is similar to that of the RS/6000 shown in Figure 3.5. This 52-bit address is used in conjunction with a 32-bit register called the table search description register 1. A 19-bit hash index is formed by taking the XOR of the 19 LSBs of the segment ID and the 16-bit page index, zero extended to 19 bits. The 9-bit mask of the TSDR1 is ANDed with the 9 MSBs of the hash output; the result is ORed with 9 LSBs of the physical address field of the TSDR1. These fields are concatenated to form the 32-bit page table address.

The page table is organized as a collection of the 64-bit (8 bytes) page table entries. The page table address indexes into a *page table entry*

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000189

| Total Main Memory | Memory for Page Tables | Number of PTEGs | Number of PTEs |
|---|---|---|---|
| 8 Mbytes | 64 Kbytes | 1 K | 8 K |
| 128 Mbytes | 1 Mbytes | 16 K | 128 K |
| 4 Gbytes | 32 Mbytes | 512 K | 4 M |

**TABLE 3.4** PowerPC 601 page table size recommendation.

*group* (PTEG) consisting of eight PTEs. The VSI + API bits of PTE0 are compared to 30 bits of the virtual address. If there is a miss, the low-order bits of the page table address are incremented by 8, to give a new PTE, and the tag of PTE1 is compared. This process continues until all eight PTEs have been tested. If there is still no hit, a second hash function is applied and the page table is again accessed. An excerpt of the recommended size of the page table is given in Table 3.4.

As with other table translation systems, the PowerPC 601 uses a translation lookaside buffer (described in Section 3.4).

*One-Level Linked List Into a Disjoint Collision Table*

J. Huck et al. [HUCK93] describes an inverted name translation system that combines a one-level system with the combined page frame address and link field of the IBM systems. This system is shown in Figure 3.26 and is called by the author a *hashed page translation table*. The first allocated entry to a hashed index is placed in the page frame table, and subsequent allocations are placed in the disjoint collision resolution table.

The page name is hashed, providing an index into the page frame table. A collision is followed by the link accessing another entry in a disjoint address space called the *collision resolution table* that cannot be accessed by a hashed index. Thus when new page information is allocated, its entry can go into (1) the page frame table if vacant or (2) any vacant slot in the collision reservation table. The entries in the collision resolution table are organized as a linked list. The advantages of this system are that noncolliding translations require only one memory reference and that the need to move entries in the page frame table is eliminated with disjoint link entry storage. As with all hashed access systems, increasing the size of the table will reduce the probability of a collision.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000190

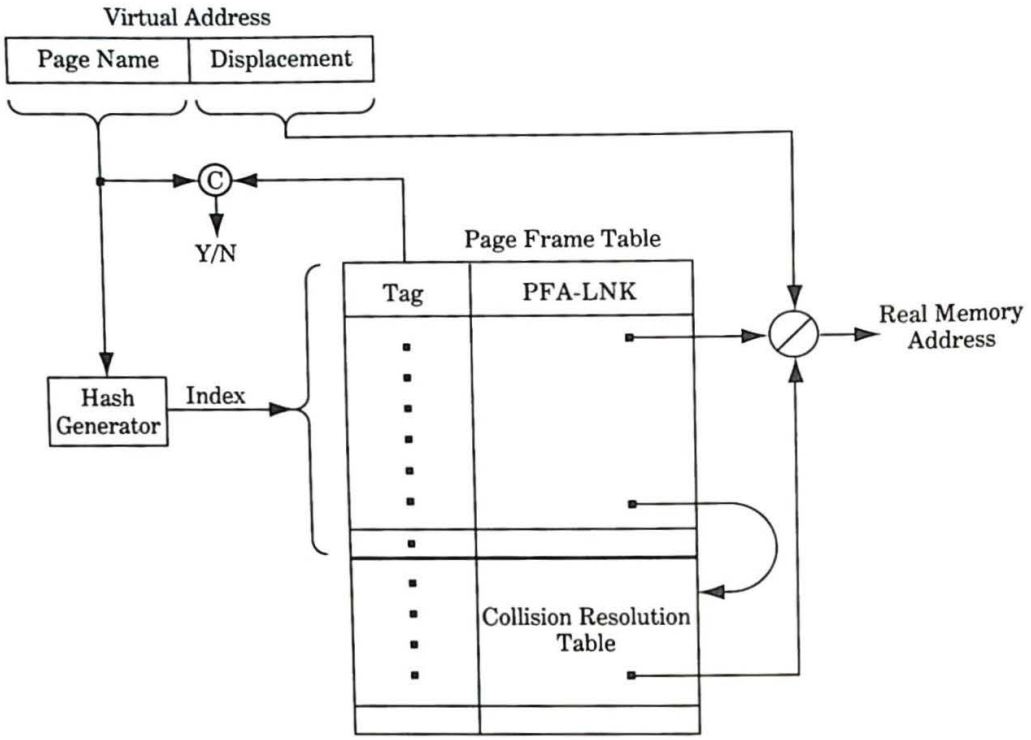



**FIGURE 3.26** Hashed page translation tables.

*Control Bits*

This section uses the IBM RS/6000 design to illustrate the control bits used in these systems. The hash anchor table has only one control bit, a valid bit that is used to indicate whether or not there is a valid pointer into the page frame table. If valid, the translation proceeds; if not, then a page fault results because an inverted page table has entries for all resident pages. In addition, there are various control and protection bits.

| | | | |
|---|---|---|---|
| i | Invalid | l | Lock Type |
| f | Reference | w | Grant Write Lock |
| c | Change | r | Grant Read Lock |
| p | Protection keys (4 bits) | a | Allow Read |
| b | Lock Bit for cache lines | | |

Other bits provide a transaction ID, and there are a number of reserved bits. Note that the RS/6000 treats valid bits nonuniformly across the hash anchor table and the page frame table. The valid bit of the hash anchor table signifies that the entry is vacant and can be allocated to a new translation. The valid bit in the page frame table signifies the presence of the requested page in real memory, assuming a hit on the page name comparison.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000191

### 3.3.2 *Segmented Systems*

Segmented systems have a variable-length allocation unit known as a *segment* that should not be confused with the fixed-length virtual address extensions shown in Table 3.1. Segment name translation can be accomplished with both one-level and multilevel systems. A one-level system is described first. The operating system sets the length of a segment under the constraint that the sum of all active segments must be no greater than the available real memory.

This discussion on segmented systems is brief because of the many problems associated with these systems. No segmented systems have been designed since the first early machines; paged-segmented systems have replaced them. Interactive programming systems such as Lotus 1-2-3 virtualize a rudimentary form of segmentation to provide interactive allocation and de-allocation of segments.

The motivation for segmentation is that the size of each allocation unit can be set to reduce internal fragmentation. With fixed size pages, some fraction of a page frame (on average, 1/2 page) will be wasted when allocating variable-length records, an issue extensively studied by Wolman [WOLM65]. Because real memory was very expensive, wasted memory was a cost that was difficult to justify and segmentation minimized this undesirable cost. However, as will be discussed, segmentation introduces a number of undesirable problems that must be solved.

The address translation process for a segmented system consists of the steps [DENN76]

$$\text{virtual address} \Rightarrow (n, d) \Rightarrow (f(b), d) \Rightarrow f(b) + d = \text{real address}.$$

That is, the virtual address is first de-allocated into the pair (segment name ($n$), displacement ($d$)). The segment name portion is translated by a function $f$, and the result is added to the displacement. Recall that for paged systems, the translated page name and displacement are concatenated because the allocation unit is fixed at some binary multiple.

The design of the processor, by selecting a displacement field length, sets the maximum length of a segment. Any allocated segment must be equal to or less than this length. Thus, the *segment length* (SL) field of the segment table will be the same length as the virtual address displacement field. When a segment is accessed, the contents of the displacement and segment length fields are compared and the access is valid if $(d) \leq (\text{SL})$.

Figure 3.27 shows a one-level segment translation table that is recognized as a direct map with an entry for every virtual segment. Because $f(n)$ is not constrained, segmented systems are not congruence mapped. That is, segments can be of variable length with the maximum length

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000192

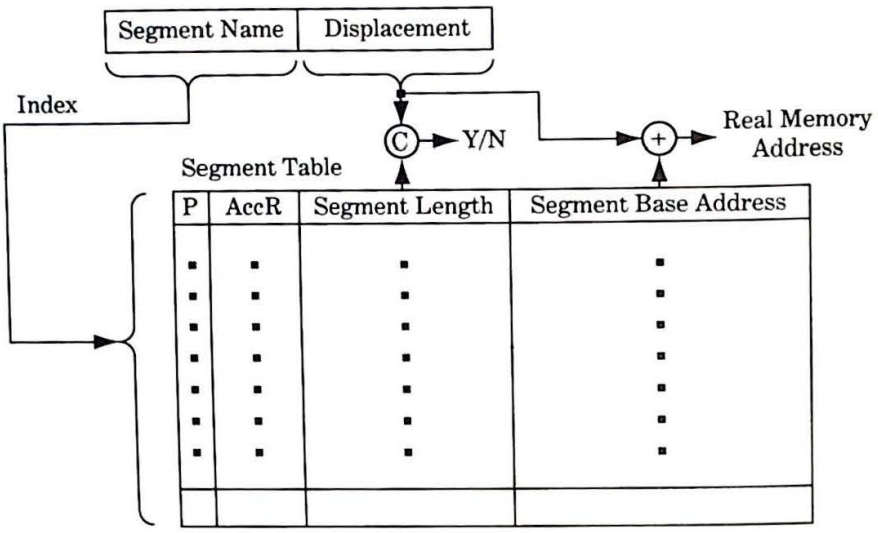



**FIGURE 3.27** One-level segment table.

set by the displacement $d$. Variable-length segments lead to the requirement that there must be a way of specifying the length in the segment table.

Segment table entries have at least the following fields:

1. present bit, similar to the present bit of a one-level direct page table,
2. access rights or control bits,
3. segment length,
4. segment base address.

The segment name indexes into the segment table. If the present bit is set, the access rights are proper, and the segment length is equal to or greater than the displacement value, then the segment base address is added to the displacement providing the real memory address. Note that because the segment base address is added to the displacement, segments can overlap, unlike paged systems. If the present bit is not set, a missing segment fault is indicated that requires a segment fetch from virtual memory. When a segment is allocated to real memory, its length is placed in the segment length field of the segment table. If the segment length is less than the displacement value or if the access rights do not check, an error signal is generated and the name translation process will not continue.

There are a number of serious problems with segmentation. For example, it can be more difficult to find a place in real memory to allocate a requested segment. This allocation problem exists because of the need to find a contiguous space in real memory that is large enough for

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000193

the new segment, a problem similar to Relocatable Partitioned Memory described in Figure 3.2.

Another problem with segmented systems is *external fragmentation*, previously defined. As segments of various sizes are moved in and out of real memory, open or unused spaces appear in real memory and the available space becomes fragmented. It becomes difficult, if not impossible, to find a contiguous space for a new segment even though there is ample, but fragmented, space in real memory. From time to time the operating system must compress or compact the allocated spaces to open up contiguous space for new segments. Note that as the allocated space is exactly the space needed, there is no internal fragmentation as with a paged system, one of the benefits of a segmented system. Section 3.5.2 discusses the allocation of segments.

I am only aware of one system that is implemented with a one-level segment table, the Rice University Computer [LEVY84]. This processor, implemented in the early 1950s, had a virtual address that specified a maximum 32-K word segment (a 15-bit displacement field). The experimental purpose of this computer was to evaluate capabilities addressing objects in conjunction with a segmented virtual memory.

### *Control Bits*

The use of the present bit was noted above. The access rights bits provide for various levels of protection. For example, read-only data segments and read/write segments can be indicated in the access rights field. Other information such as lexical level and the data type of the segment can also be indicated.

### *Multilevel Segmentation Systems*

A multilevel segmented system uses the first-level table to select subsequent translations for each of the active processes as shown in Figure 3.28. Note that the first level is indexed by the MSBs of the segment name and organized like a paged system with the first level similar to the map shown in Figure 3.5. The second level is identical to that shown in Figure 3.27.

The LSBs of the segment name and the table address are concatenated to form the segment table address, thus segment overlap is prevented. Said another way, there are $2^B$ nonoverlapping segment tables of $2^{\text{Table Address}}$ entries. I know of no actual system that is implemented with multilevel segment tables. Concurrent with the realization of the problems with pure segmentation, paged segmentation became the implementation method of choice and is discussed in the following section.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000194

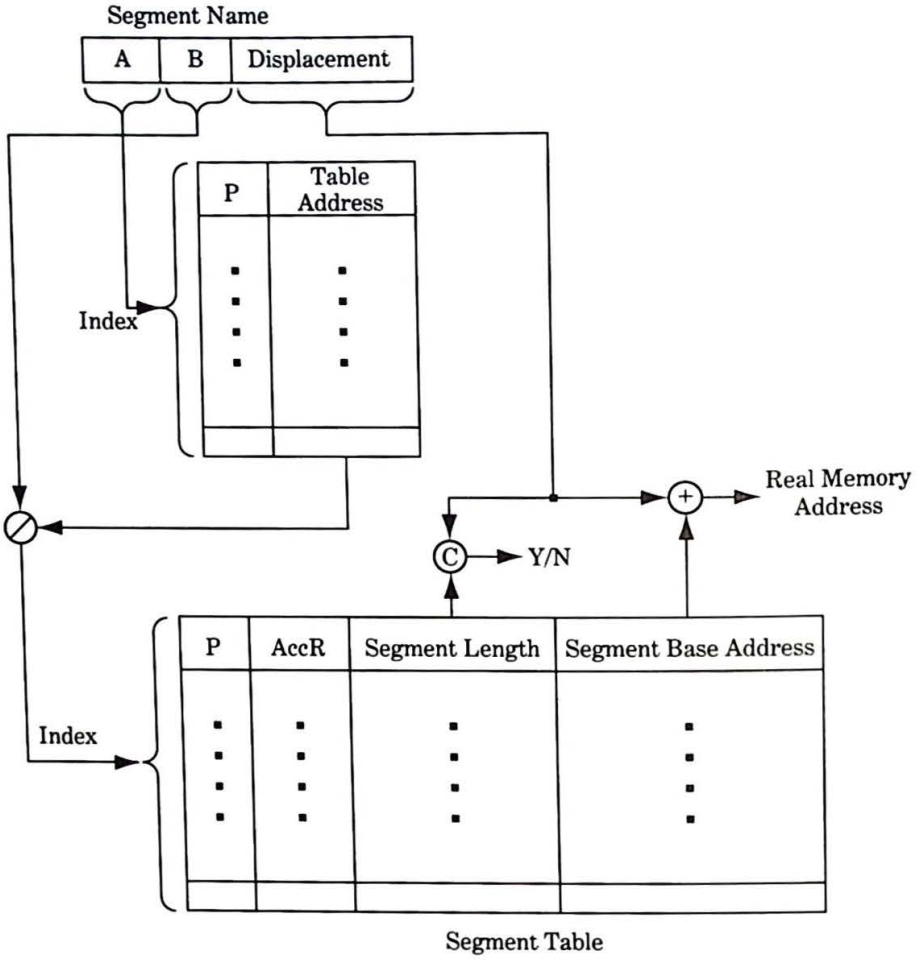



**FIGURE 3.28** Two-level segment table.

### 3.3.3 *Paged Segmentation*

As noted previously, there are a number of problems with segmentation systems that have hampered their use. One significant problem involves finding space for a new segment in real memory. This allocation problem is difficult, and external fragmentation can significantly reduce the effectiveness of real memory. If compaction is used, there can be a serious reduction in performance. Another significant problem is that a TLB, described in Section 3.5, has to be allocated at the word or byte level; this process makes the TLB quite large, slow, and expensive. Researchers have recognized that if the allocation unit is a page of, say, 1 Kbyte, a compromise system combining the characteristics of a segment system

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000195

with a paged system could be designed. In other words, a segment is not a variable number of AUs but is a variable number of pages.

The tradeoffs that favor a paged-segmented system over a pure segmented system concern accepting internal fragmentation for no external fragmentation, the simplicity of allocating a page as compared to a variable-length segment, and the performance improvement due to simpler allocation.

The organization of a paged-segmentation system is shown in Figure 3.29. This system has two levels; the first level establishes the segments of pages while the second level is a one-level page system. Note that the second level could translate page names by any one of the techniques discussed above. This translation can be associatively searched, one-level or multilevel page tables or translation information could be hash addressed. The example shown, however, is a one-level page translation system.

The virtual address is divided into three fields: segment name, page name, and displacement. The segment name may be concatenated with a user ID number, forming an index into the segment table. The register holding the user ID is also known as *segment table origin register*. A length field in the segment table is compared with the virtual page name

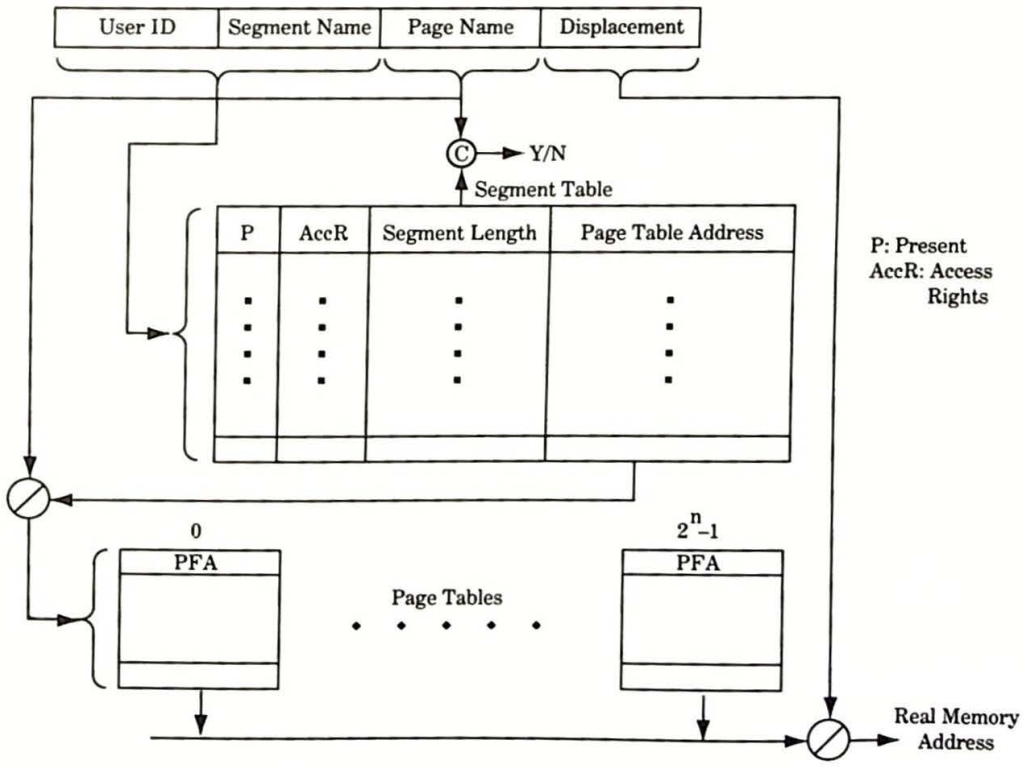



**FIGURE 3.29** Paged segmentation.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000196

field to verify that the length of the segment, in increments of pages, is proper. In other words, the page name is a displacement into a segment. The page table address, from the segment table, is concatenated to the page name field to index into the page tables. The selected page frame address is concatenated with the displacement field to give the real memory address. A present bit indicates whether or not an entry in the segment table has its corresponding page table in real memory. Access rights bits provide for protection and control.

Only multilevel paged-segmentation systems are possible; one-level systems cannot be implemented. This is because, as shown in Figure 3.29, at least one level is required for the segmentation table and another for the page tables. As with the other systems discussed earlier, all of the page tables do not need to be resident in real memory at once, only the active tables. If an access is made to a segment or page table that is not in memory, it can be assumed that the page is not in memory and the operating system must fetch the page(s) and update the segment/age tables. The Multics system paged the segment table, leading to complications discussed below.

The paged-segmented system overcomes many of the problems of a pure segmented system. The unit of allocation of a segmented system is a word or byte, while the unit of allocation of a paged system is a page. Because of this difference, the segment system can now operate as a demand paged system that shifts the problem of finding space to the page system. Also, the problem of external fragmentation and space compaction found in pure segmented systems are eliminated. However, because the pages of a segment are not placed in contiguous locations of real memory, locality for caching may be degraded; the same effect is found with a paged system that does not consider locality during a page swap.

For the above reasons, many of the high-performance microprocessor memory management units now support both paged and paged-segmented operation [MILE90]. The operating system can manage a virtual segment table on top of a pure paged system. The page tables are stored in memory, loaded and modified by the operating system, and may be paged themselves. See [MOTO87] for the details of a system. With such a system, no special hardware is needed to support the segment table as a paged-segmented system can be implemented in software on a hardware-paged system. Thus the two systems are isomorphic.

### Segment Table Support

Because of the complicated organization of segment tables, interpretive segmentation has been found to be out of the question for performance reasons. Therefore, hardware support is mandatory if a reasonable level

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000197

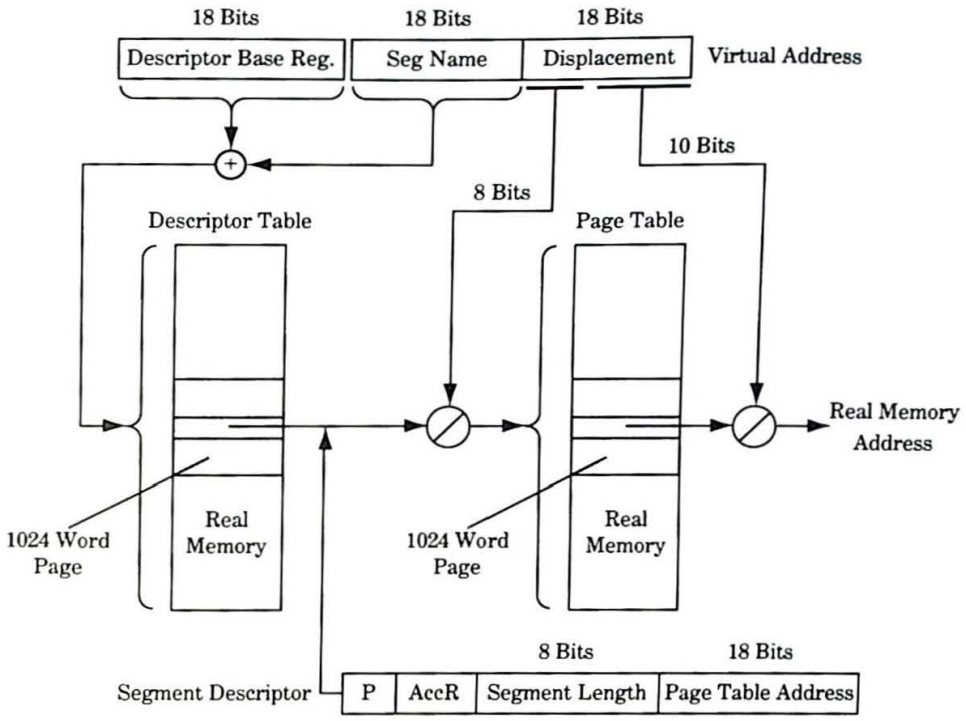



**FIGURE 3.30** Multics name translation tables.

of performance is to be achieved. One of the earliest segmentation systems is Multics, which was developed at MIT in conjunction with General Electric. A GE 635 computer was modified to support paged segmentation becoming the GE 645. Multics system [SCHR71, ORGA72] used a multilevel paged-segmentation system completely implemented in software, augmented with a TLB. Figure 3.30 shows a simplified view of the Multics segment name translation system.

The processor produces a 36-bit effective address, the page size is 1K Words, and a segment can have up to 256 pages or 256K Words. The effective address is augmented with an 18-bit program loaded *descriptor base register* that points to the beginning of a segment in the descriptor table allocation of real memory that contains *segment descriptors*. Note that the segment descriptors are themselves paged.

A segment descriptor contains a segment length field of 8 bits, a page table address of 18 bits, and various control bits. The segment length field is compared, by the processor, to the eight MSBs of the displacement to verify that a displacement is within the allocated segment. The page table address is concatenated with the eight MSBs of the displacement to form the address into the page table, also in real memory and paged. The entries in the page table contain the page frame

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1022, Page 000198