TABLE 4.8 INSTRUCTION CACHE P_{miss} SPEC BENCHMARKS, OVERALL

	Number of Sets						
n-Way	64	128	256	512	1K	2K	4K
1, Direct	0.051	0.039	0.031	*0.020*	0.012	0.007	0.003
2	0.034	0.024	*0.017*	0.010	0.008	0.003	0.0002
4	0.023	*0.016*	0.009	0.008	0.001	0.0002	0.0001
8	*0.016*	0.009	0.008	0.0007	0.0002	0.000	0.000

16-BYTE BLOCK, ONE BLOCK SLOT PER SET

Hit data and miss data for instruction caches can be found in a number of references (Gee et al. 1993). An abstracted set of miss data for SPEC benchmarks and a 16-byte block is shown in Table 4.8. "Overall" means the complete set of integer and floating-point benchmarks.

This table is organized so that caches of equal size are on the diagonal (lower left to upper right). For example, an 8K byte cache, shown in italics, can be organized as eight-way set associative with 64 sets. This cache can also be organized as a direct cache with 512 sets. Note that in general the organization of the cache has little influence on P_{miss}; size alone is the major determinant. For small instruction caches, there is a slight reduction in P_{miss} as the degree of associativity is increased and the number of sets decreases.

Instruction caches also support temporal locality or look behind. Particularly for scientific programs with looping, the cache should be large enough to hold a loop. After first fetching the loop, it is in the cache and will not need to be fetched again.

In the evolution of computer implementation hardware, queues have been used to perform the look ahead function while instruction buffers have been used for the look behind function (Milligan and Cragon 1995). By their design, caches perform both functions. Queues are used in conjunction with caches on a number of modern processors such as the Pentium Pro and the PowerPC.

Data Caches

Data references generally show little spatial locality,[46] but when a datum is once used it is frequently used again; this is known as temporal locality. The data cache memory saves previously referenced data items in anticipation of future use. Miss data for SPEC benchmarks are shown in Table 4.9 for a 16-byte block. Caches of equal size are shown on the diagonal.

As with instruction caches, the major determinant of P_{miss} is the size of the cache. For small caches the miss rate decreases with an increase in set associativity. However,

[46] Scientific programs with array data types are an exception to this statement. Spatial locality is so good that some supercomputers do not use a data cache.

Petitioner Exhibit 1053, Page 168 of 334

TABLE 4.9 DATA CACHE P_{miss}, SPEC BENCHMARKS, OVERALL

	Number of Sets						
n-Way	64	128	256	512	1K	2K	4K
1, Direct	0.255	0.216	0.170	0.114	0.087	0.069	0.057
2	0.188	0.155	0.104	0.074	0.059	0.048	0.039
4	0.134	0.098	0.071	0.058	0.047	0.038	0.028
8	0.091	0.069	0.057	0.047	0.038	0.028	0.017

16-BYTE BLOCK, ONE BLOCK SLOT PER SET

note that a direct cache has a higher miss rate than two-, four-, or eight-way set associative caches of the same size. Direct caches have a shorter logic path, by a few gates, than the other caches. Thus it is possible, but not proven, that with a reduced clock period that is due to the shorter logic path, a cache with shorter latency can be effective even though the P_{miss} is greater. With a faster cache the processor can have a faster clock, resulting in an overall speed improvement for the computer.

Unified Caches

Unified caches store both instructions and data and are used today as a second-level cache to the on-chip instruction and data caches. As would be expected, the P_{miss} of a unified cache is between an instruction cache and a data cache. Table 4.10 shows P_{miss} data for the same SPEC benchmarks and 16-byte block.

As with instruction and data caches, the size of a unified cache is the important parameter for establishing a low P_{miss}. However, for a given size unified cache, the P_{miss} of a direct cache is approximately 20% to 25% greater than that of a two-way set associative cache. Thus second-level unified caches are generally implemented as two-way or eight-way set associative.

TABLE 4.10 UNIFIED CACHE P_{miss}, SPEC BENCHMARKS, OVERALL

	Number of Sets						
n-Way	64	128	256	512	1K	2K	4K
1, Direct	0.151	0.114	0.086	0.056	0.039	0.029	0.020
2	0.092	0.069	0.047	0.031	0.023	0.016	0.011
4	0.062	0.044	0.029	0.022	0.015	0.010	0.007
8	0.041	0.028	0.021	0.015	0.010	0.008	0.005

16-BYTE BLOCK, ONE BLOCK SLOT PER SET

Petitioner Exhibit 1053, Page 169 of 334

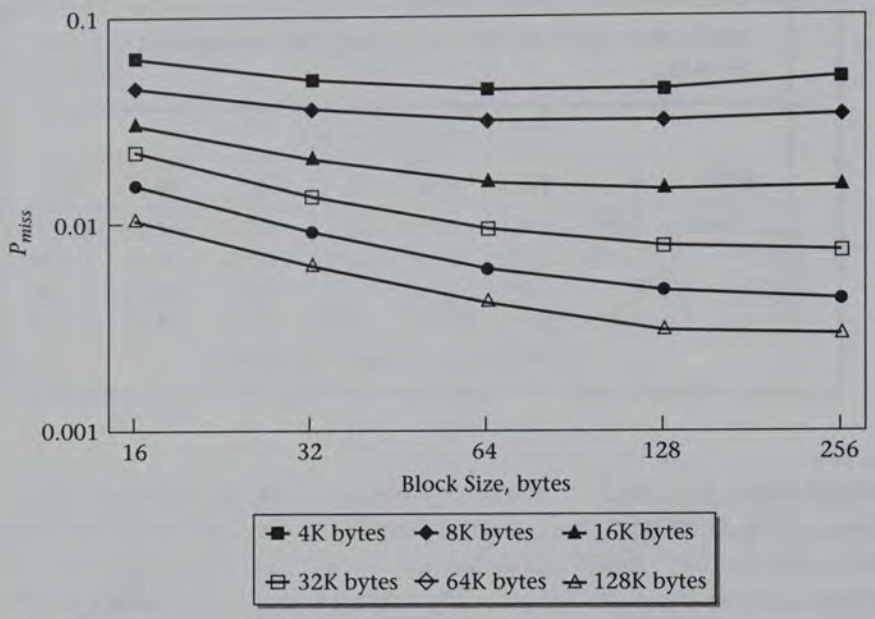


Figure 4.18 P_{miss}, block size and cache size

Block Size

The P_{miss} data presented in Tables 4.8– 4.10 are for a 16-byte block. For the designer, what determines the size of the block? There are two considerations: (1) how does block size affect P_{miss}, and (2) how does block size affect the real-memory-to-cache transport time?

For the issue of P_{miss} and block size, one can look at two extremes: a block of 1 AU at one extreme and a block the size of the cache at the other extreme. For the small block, there will be no look ahead and for the very large block there is little look behind. In both cases, the P_{miss} will be large. Figure 4.18 shows the P_{miss} for various cache sizes and block sizes for a unified cache (Gee et al. 1993). These data are for a unified cache, four-way associative organization, and combined integer and floating-point SPEC benchmarks.

Note that for small caches there is a pronounced minimum P_{miss} as the block size increases. As the cache size increases, the minimum moves out to larger block sizes. For large caches, such as the 64K-byte and the 128K-byte caches, there is a relatively constant reduction in P_{miss} with each doubling of block size – a decrease with a model that is similar to the learning curve model described in Chapter 2:

$$P_{miss}(\text{final}) = P_{miss}(\text{initial}) L^{\log 2 \frac{\text{final block size}}{\text{initial block size}}}$$

The 64K-byte cache shown in Figure 4.18 has $L \approx 0.7$. In other words, as the block size doubles, the P_{miss} is approximately 0.7 the previous P_{miss}. The issue of the effect of block size on transport time is discussed in Section 4.4 in which the bus interface between the cache and the real memory is modeled.

Petitioner Exhibit 1053, Page 170 of 334

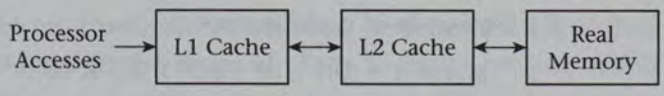


Figure 4.19 Two-level cache

Multilevel Caches

Many processors today have an instruction cache and a data cache on the processor chip with a second-level cache off-chip. A block diagram of a two-level cache is shown in Figure 4.19. The caches are commonly known as L1 and L2 caches.

For a one-level cache, the ratio of the transport time T_{rc} to the L1 cache latency can be in the range of 10–50. There are a number of reasons for this high ratio: The real memory is on a bus and there is bus synchronization time, and a block requires a number of clocks for its transfer across a bus that may be only a few bytes wide. For this reason, a second-level L2 cache is used to reduce the effect of large values of T_{rc}.

A model for the effective latency of a two-level cache t_{elc2} can be derived as follows. The L1 cache has a latency of 1 cycle, the L2 to L1 transport time is T_{c2}, and the real-memory transport time to the L2 cache is T_{rc3}. Transport times T_{c2} and T_{rc3} are in cache cycles. P_{miss1} and P_{miss2} are the miss rates of the L1 and the L2 caches, respectively. The model assumes that all references are satisfied in one of the three levels of memory.

When there is a hit on the L1 cache, the L2 cache and the real memory are "don't-care" conditions. Likewise, with a miss on the L1 cache and a hit on the L2 cache, the real memory is a don't-care condition. Table 4.11 shows the derivation of the effective latency model.

$$\begin{aligned} t_{elc2} &= 1 - P_{miss1} + P_{miss1}(1 - P_{miss2})(1 + T_{c2}) + P_{miss1}P_{miss2}(1 + T_{c2} + T_{rc3}) \\ &= 1 + P_{miss1}T_{c2} + P_{miss1}P_{miss2}T_{rc3} \end{aligned}$$

EXAMPLE

A one-level cache system has $P_{miss} = 0.1$ and $T_{rc} = 10$. Consider a two-level cache memory with $P_{miss1} = 0.1$, $P_{miss2} = 0.1$, $T_{c2} = 3$, and $T_{rc3} = 10$. What is the speedup in effective latency with the addition of the L2 cache?

Solution

The effective latency of the single-level cache system is

$$t_{elc} = 1 + 0.1 \times 10 = 2 \text{ clocks.}$$

TABLE 4.11 MULTILEVEL CACHE EFFECTIVE LATENCY

L1	*L2*	RM	Probability	Time	Product
Hit	x	x	$1 - P_{miss1}$	1	$1 - P_{miss1}$
Miss	Hit	x	$P_{miss1}(1 - P_{miss2})$	$1 + T_{c2}$	$P_{miss1}(1 - P_{miss2})(1 + T_{c2})$
Miss	Miss	Hit	$P_{miss1}P_{miss2}$	$1 + T_{c2} + T_{rc3}$	$P_{miss1}P_{miss2}(1 + T_{c2} + T_{rc3})$
			SUM = 1		SUM = t_{elc2}

For the two-level cache memory,

$$t_{elc2} = 1 + (0.1 \times 3) + (0.1 \times 0.1 \times 10)$$
$$= 1.4 \text{ clocks.}$$
$$\text{speedup} = 2/1.4 = 1.43.$$

Comments

By the introduction of an L2 cache that has a smaller T_c than the T_{rc} of the one-level cache, the cache latency has been significantly improved by 30%. An L2 cache will be beneficial only if t_{c2} is significantly less than the transport time between a single-level cache and real memory.

The two-level cache memory model developed above can be extended to any number of levels of cache or extended to comprehend the virtual-memory level. The P_{miss} values used in this model are called local miss probabilities; that is, the probability of a request entering a cache level and missing. Another approach to multilevel cache modeling is to use global hit probabilities. A global hit probability is the fraction of total references that are served at a given level; global hit probabilities must add to 1. The global hit probabilities for the example above are

$$P_{hit1g} = 0.9,$$
$$P_{hit2g} = 0.1 \times 0.9 = 0.09,$$
$$P_{hit3g} = 1 - 0.9 - 0.09 = 0.01.$$

EXAMPLE

Rework the example above for the two-level cache by using global hit rates.

Solution

$$t_{elc} = (0.9 \times 1) + [0.1(1 + 10)] = 2,$$
$$t_{elc2} = (0.9 \times 1) + [0.09(1 + 3)] + [0.01(1 + 3 + 10)]$$
$$= 0.9 + 0.36 + 0.14 = 1.4,$$
$$\text{speedup} = 2/1.4 = 1.43$$

A block diagram of a modern memory design is shown in Figure 4.20. This memory has two L1 caches, an instruction and data cache on the chip, with an off-chip unified L2 cache.

Because of the very close coupling between the processor and the on-chip caches, the transport time between the instruction/data caches and the unified cache can be

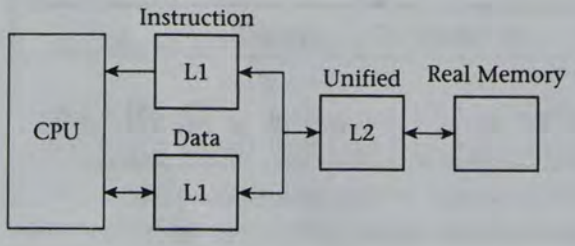


Figure 4.20 Multilevel cache

Petitioner Exhibit 1053, Page 172 of 334

small – in the range of 2 to 4 cycles. However, the transport time between the unified cache and the real memory may be in the range of 10 to 50 cycles because of bus synchronization requirements.

EXAMPLE

The instruction and data caches of Figure 4.20 are both 8K bytes in size and two-way set associative. The unified cache is 32K bytes in size and two-way set associative. Assume a 16-byte block and a local P_{miss} of real memory of zero; that is, virtual memory is not used.

a. What fraction of instruction and data references are found in real memory?

b. If each instruction has 0.3 data references, what is the weighted average number of memory references per instruction that are satisfied in real memory?

c. Evaluate (b) under the assumption that there is no second level cache.

Solution

a. From Tables 4.8, 4.9, and 4.10 we find that the instruction cache P_{miss} is 0.017, the data cache P_{miss} is 0.104, and the unified cache P_{miss} is 0.023. The fraction of instruction references that are found in real memory is

$$0.017 \times 0.023 = 0.00039.$$

The fraction of data references that are found in real memory is

$$0.104 \times 0.023 = 0.00239.$$

b. The weighted average number of memory references per instruction that are satisfied in real memory is

$$1 \times 0.00039 + 0.3 \times 0.00239 = 0.0011.$$

c. Real-memory references per instruction without second-level cache are

$$1 \times 0.017 + 0.3 \times 0.104 = 0.048.$$

Comments

Using a second-level cache reduces the fraction of references per instruction to real memory from approximately 0.05 to approximately 0.001 per instruction. This reduces the loading on the bus and permits more concurrent I/O operations to proceed without taking too many cycles from the processor.

Special Caches

There are a number of special buffers in a processor that are cachelike in their organization and use. In general these special caches are used to take advantage of temporal locality.

A TLB is one of these special caches, mentioned briefly in Section 4.2. The TLB holds translated page names that have been translated by the page tables on a prior reference. The TLB of the Intel Pentium Pro is shown in Figure 4.21. There are two TLBs, one for instructions and one for data.

This TLB is four-way set associative and has only eight sets; a valid bit is associated with each entry. Translated addresses are stored in this cache instead of data or instructions. There is little published information on the miss rate of TLBs, but what there is

Petitioner Exhibit 1053, Page 173 of 334

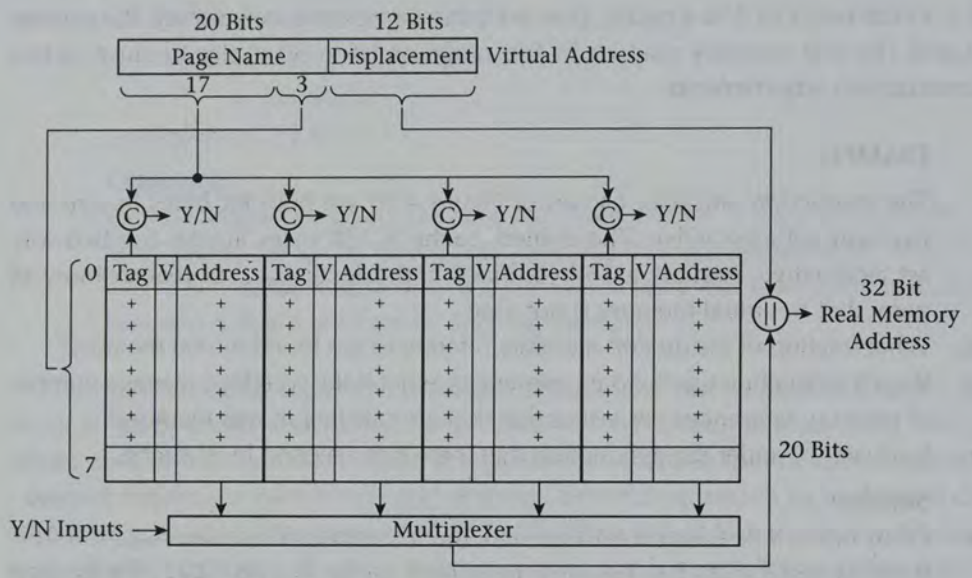


Figure 4.21 Intel Pentium Pro TLB (courtesy Jones and Bartlett Publishers)

(Saavedra and Smith 1993) suggests that $P_{miss} = 0.01 - 0.02$. However, this miss rate is not as bad as it sounds.

Instruction and data references are made to the caches, and the TLB is usually accessed in parallel with the cache access, as shown in Figure 4.22. Thus, if the cache access is a miss, the translated address is already available, assuming there is not a TLB miss. With a cache $P_{miss} = 0.1$, the number of memory references that need translation is 10%. With the TLB $P_{miss} = 0.01$, the number of references that must be translated in the page tables is $0.1 \times 0.01 = 0.001$ or 1 in 100. This ratio is even smaller when a second-level cache is used.

EXAMPLE

A cache has $P_{missC} = 0.1$, and a TLB has $P_{missTLB} = 0.01$. The latency of the cache is 1, the latency TLB is 5% of the real-memory cycle time, the table translation takes 2 memory cycles, and a memory cycle is 4 cache cycles. What is the effective real-memory latency?

Solution

The TLB is searched in parallel with the cache; its contents are used only if there is a cache miss and it is faster than the cache. Thus the TLB latency does not

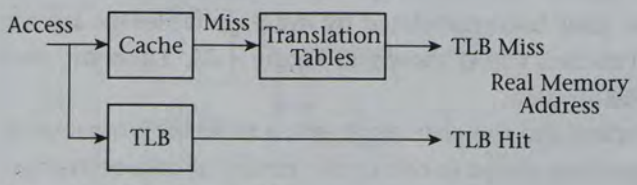


Figure 4.22 Cache and TLB access

Petitioner Exhibit 1053, Page 174 of 334

TABLE 4.12 EFFECTIVE REAL-MEMORY LATENCY

Event	Probability	Time	Product
Hit cache	$1 - P_{missC}$	1	$1 - P_{missC}$
Miss cache, hit on TLB	$P_{missC} \times (1 - P_{miss\ TLB})$	$0.05 \times 4 + 4$	$4.2 P_{missC} \times (1 - P_{miss\ TLB})$
Miss cache, miss on TLB	$P_{missC} \times P_{miss\ TLB}$	$1 + (2 + 1)4$	$13 P_{missC} \times P_{miss\ TLB}$
			SUM = EFFECTIVE REAL-MEMORY LATENCY

add to the access time on a TLB hit. Table 4.12 shows events, probabilities, and times needed to solve this example.

The effective real-memory latency is

$$\begin{aligned} & 1 - P_{missC} + 4.2 P_{missC} \times (1 - P_{missTLB}) + 13 P_{missC} \times P_{missTLB} \\ & = (1 - 0.1) + (4.2 \times 0.1 \times 0.99) + (13 \times 0.1 \times 0.01) \\ & = 1.33 \text{ cycles.} \end{aligned}$$

Comments

This example shows that page table translation by a number of levels does not significantly increase the latency of real memory if a TLB is used.

Although the organization of the Intel Pentium Pro TLB is eight-way set associative, other processors use other organizations. For example, the IBM S370 uses an inverted TLB that is accessed by hashing the page name (similar to the inverted page tables discussed in Section 4.2).

A TLB has a read policy similar to that of an instruction cache. When there is a TLB miss, after the translation is performed in the page tables, the TLB is updated. As a TLB is not written by the processor, unlike a data cache, the new translation information can be overwritten into the TLB.[47] Needless to say, the TLB has no write policy. In some systems, the TLB is switched out/in on a context switch. Switching saves the time required for making page name translations when the old program is restarted.

Some TLBs have some locations hardwired to a given page frame address, usually the low address pages. The reason for this is to have an identity mapping for operating system pages. We do not want the system to have page faults when it is trying to handle a page fault.

4.4 INTERLEAVED REAL MEMORY

The real memory between the cache and the disk requires special properties. This memory is accessed only when there is a read or write miss by the cache. When such an

[47] A TLB is updated as a side effect of a page name translation by means of the page tables.

Petitioner Exhibit 1053, Page 175 of 334

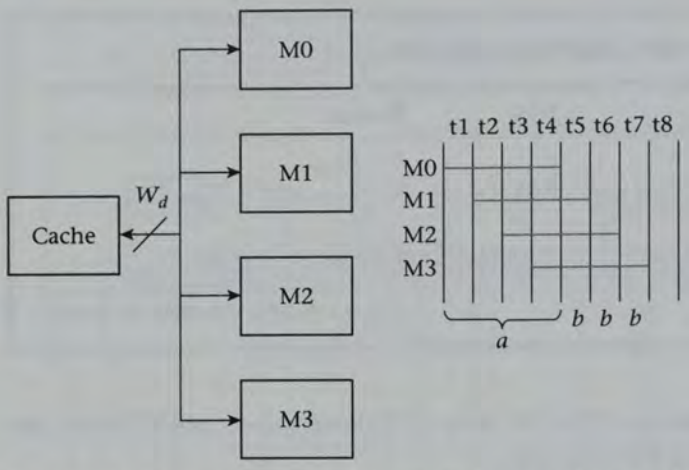


Figure 4.23 Four-way interleaved memory (courtesy Jones and Bartlett Publishers)

event occurs, it is necessary to transfer a block to/from the cache. Today, the technology used for real memory is DRAM with a latency of 10–50 cache cycles. Fortunately, a block is composed of a number of AUs; thus the bandwidth of the real memory is as important as its latency. How can we provide high bandwidth with these relatively slow memory components? The answer is found in the use of parallel memory modules called interleaving.

Figure 4.23 shows a four-way interleaved memory. Four memory modules with word-length W_d AUs are connected to a bus having width W_d AUs, and the bus is connected to the cache. The clock period of the bus is the same as the cache cycle and sets the clock period of the system. For this example, the cycle time of the memory modules is divided into four clocks. The first module, M0, is accessed, and one clock period later, M1 is accessed. This continues until all the modules have been accessed, and the process can repeat. The output (assuming a read) of M0 is available at the end of t4, of M1 at t5, and so on. The configuration is sometimes known as *C* access because the modules are accessed concurrently, although skewed.

The transport time T_{rc} is the time to transport a block to/from the real memory and the cache. For interleaved memories, T_{rc} in bus clocks is

$$T_{rc} = a + b\left(\frac{A}{W_d} - 1\right),$$

where a is the cycle time of the memory modules in bus clocks, the time to produce the first memory output, and the interleaving factor. The number of clocks required for producing each subsequent output is b clocks; usually equal to 1. A is the number of AUs in a block, and W_d is the width of the bus in AUs.

The transport time model is now modified to include bus synchronization time, which is discussed further in Chapter 7. Assume that the bus synchronization requires s cycles. The transport time becomes

$$T_{rc} = s + a + b\left(\frac{A}{W_d} - 1\right) \text{ clocks.}$$

Petitioner Exhibit 1053, Page 176 of 334

EXAMPLE

Consider an interleaved memory having the following characteristics. The interleaving factor is 4, $s = 3$, $A = 32$ bytes, and $W_d = 4$ bytes. What is the block transport time?

Solution

$$T_{rc} = 3 + 4 + \left(\frac{32}{4} - 1\right) = 15 \text{ clocks.}$$

Comment

As $A \to \infty$, $T_{rc} \to A/W_d$. In other words, the time to synchronize the bus and read the first module is prorated over a large number of bytes transferred.

With this model we revisit the issue of block size. A model of the effective cache latency is derived by using the model for T_{rc} and the model for P_{miss} in the basic model for cache effective latency t_{elc}:

$$t_{elc}(\text{in clocks}) = 1 + P_{miss} T_{rc},$$

$$P_{miss}(\text{final}) = P_{miss}(\text{initial}) L^{\log 2 \frac{\text{final block size}}{\text{initial block size}}},$$

$$T_{rc} = s + a + b\left(\frac{A}{W_d} - 1\right) \text{clocks},$$

$$t_{elc}(\text{in clocks}) = 1 + P_{miss}(\text{initial})\, L^{\log 2 \frac{\text{final block size}}{\text{initial block size}}} \left[s + a + b\left(\frac{A}{W_d} - 1\right)\right].$$

Figure 4.24 shows, from this model, how the effective cache latency changes as the block size increases from 16 to 256 bytes. The memory parameters used in the model are $s = 4, a = 4, b = 1$, and $W_d = 8$. Cache parameters, from Figure 4.18, are for an 8K-byte unified four-way set associative cache. Initial $P_{\text{miss}} = 0.015$ for a 16-byte block and $L = 0.7$.

The significant result of Figure 4.24 is that there is an optimum block size with other memory system parameters fixed. The shallow minimum occurs, for this design, at 64 bytes. With larger block sizes, transport time overwhelms the reduction in P_{miss} and

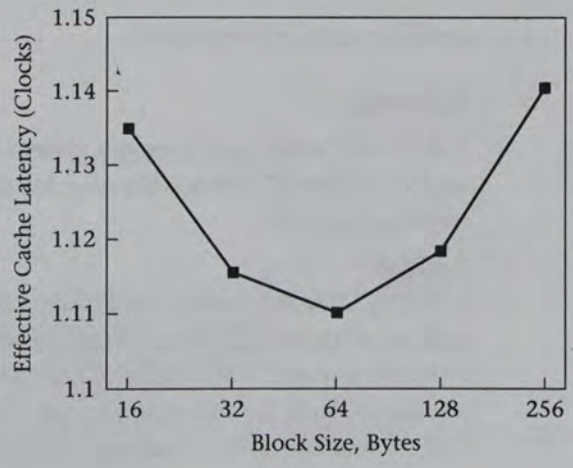


Figure 4.24 Effective cache latency vs. block size

Petitioner Exhibit 1053, Page 177 of 334

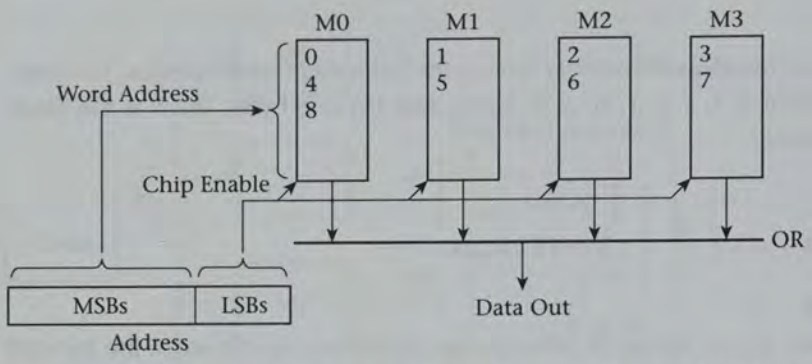


Figure 4.25 Low-order bit interleaving (courtesy Jones and Bartlett Publishers)

the effective cache latency begins to increase. Because of the shallow minimum, other design considerations may shift the block size up or down (32 or 128 bytes) from the optimum value.

This model can be used to evaluate the change in cache latency with a change in other parameters such as cache size, bus width, real-memory interleaving factor, and bus synchronization.

Interleaved Memory Addressing

A method for addressing an interleaved memory is shown in Figure 4.25 for four-way interleaving. The address is divided into two fields; the two LSBs are decoded and provide chip enable signals to each of the four modules. The MSBs are decoded and select the same row from each module. As the LSBs increment, as in reading a block of consecutive AUs, the outputs are OR'd together to provide the data out to the bus.

Note how the addresses are allocated to memory. Module M0 has addresses 0, 4, 8, etc. Module M1 has addresses 1, 5, 9, etc. As long as the address increment is 1, the data out rate will be one AU per clock. However, if the address increment is 2, the data out rate will be one AU per 2 clocks. If the increment is 4, the data out rate will be one AU per 4 clocks. In other words, the speedup of an interleaved memory system over a single memory module is

$$1 \leq \text{speedup} \leq \text{interleaving factor}.$$

EXAMPLE

A four-way interleaved memory system is accessed by an address stream with an increment of 2. What is the speedup of this interleaved memory over a single memory module?

Solution

A timing diagram, similar to Figure 4.23, can be sketched for this problem, shown in Figure 4.26. Five clocks are required for reading the first pair of AUs from M0 and M2. The next read from M0 and M2 takes four additional clocks, and so on. Thus, for this example, the number of clocks per AU is $2n+1$, where n is the number of AUs read from the memory and n is even.

Petitioner Exhibit 1053, Page 178 of 334

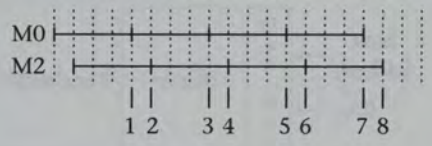


Figure 4.26 Interleaving example

For one memory module, the number of clocks per AU is *na* clocks, where *a* is the interleaving factor. Thus the speedup of the interleaved memory over a single module is

$$\text{speedup} = \frac{na}{2n+1}.$$

For large *n*, the speedup is $(4/2) = 2$.

Wide-Word Memory

Another technique for increasing the bandwidth of a memory system is to use a wide-word memory. With this technique, a single memory cycle places a wide word consisting of a number of AUs into a buffer that is then unloaded onto the bus. Figure 4.25 is redrawn in Figure 4.27 to illustrate this technique. The address MSBs access all of the memory modules at one time. Their outputs are simultaneously placed in a buffer. The address LSBs then sequence the data from the buffer onto the data out bus.

This interleaved memory configuration is sometimes called *S* access because the modules are accessed simultaneously. The speedup of this configuration is identical to that of the *C* access configuration as shown in Figure 4.23. This configuration has the disadvantage of electrical noise problems as all of the memory module switching transients occur at the same time.

MOS DRAM Organizations

The invention of the MOS DRAM started a major revolution in computer memory design. The first DRAM organization is shown in Figure 4.28. The memory is usually a square array of one-transistor cells. The address MSBs (row address) access the rows and the address LSBs (column address) access one column. A read or write operation involves a single bit on the I/O pin.

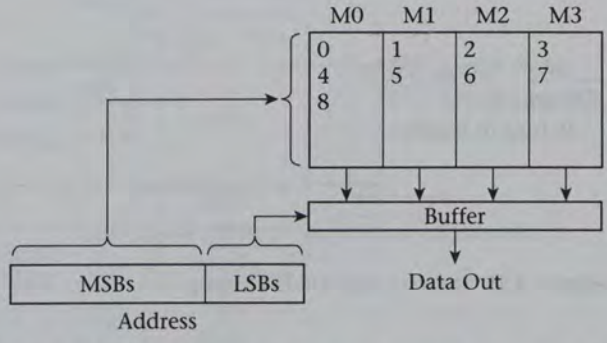


Figure 4.27 Wide-word memory (courtesy Jones and Bartlett Publishers)

Petitioner Exhibit 1053, Page 179 of 334

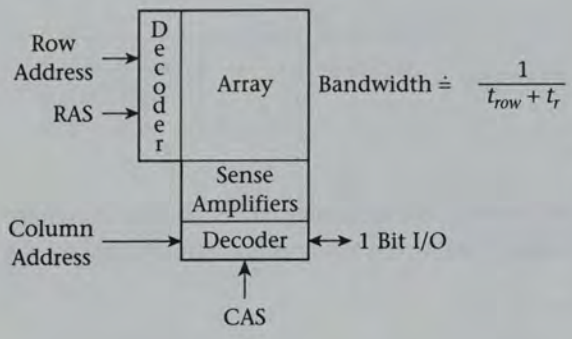


Figure 4.28 Basic DRAM Organization

An approximate and stylized timing diagram is shown in Figure 4.29 for the basic DRAM organization. The row access consisting of the row address and the row address strobe (RAS) starts a read operation. The column access starts after the row access. In general, the row access time is longer than the column access time. For $t_{row} > t_{col}$, the access time is $\approx t_{row}$. After the array is accessed, an additional time t_r is required for returning the various signals to their inactive state. Thus the cycle time is the sum $t_{row} + t_r$.[48] The bandwidth of this organization is one bit per cycle.

Cragon (Cragon 1980) pointed out that as the number of bits accessed in a row is the square root of the array bits, a significant portion of the potential bandwidth of the memory is wasted. For example, a 64K-bit array will access 256 bits in a single row access but only one bit is used. To overcome this waste in bandwidth, there have been a series of improvements to the organization to use the latent bandwidth more effectively.

The following paragraphs discuss only three of the multitude of DRAM organizations that have been produced. For every organization, the design goal has been to reduce the latency (random-access time) and to increase the bandwidth. Please note that different manufacturers use different names and terminology for describing their DRAMs.

These DRAMs operate asynchronously. That is, a controller produces various timing signals to the DRAM chips that sequence the row and column access. The output appears on the I/O pins in some guaranteed time, and it is the responsibility of the memory system designer to see that the outputs are latched into a register. This register can then be read with a system synchronous clock.

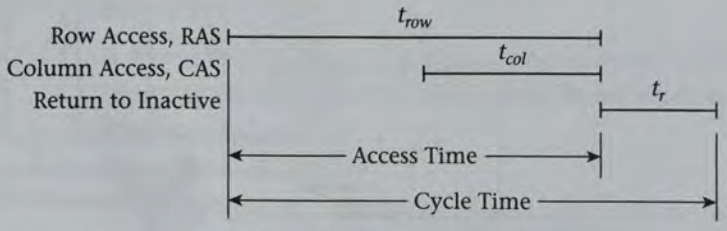


Figure 4.29 Stylized basic DRAM timing

[48] There are a large number of other timing parameters, but this simple model permits relative performance evaluation of the various DRAM organizations.

Petitioner Exhibit 1053, Page 180 of 334

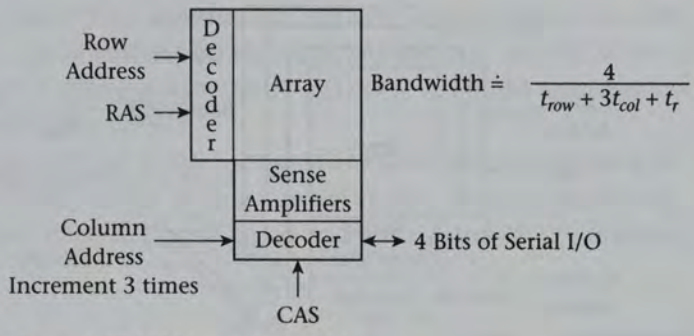


Figure 4.30 Nibble-mode DRAM

Nibble-Mode DRAM. An early variant of the basic DRAM is the nibble-mode DRAM, shown in Figure 4.30. This organization performs a read row access, then four column accesses that place a stream of 4 bits on the I/O pin. The first bit requires t_{row}. Each additional bit requires another column access cycle t_{col}; then t_r is required to complete the operation. Nibble mode provides approximately 90% increase in bandwidth over the basic DRAM organization.

Page-Mode DRAM. The number of bits in a DRAM doubled approximately every 1.5 years over the period from 1978 to 1996, an increase greater than that forecast by Moore's law. As the number of bits in the array increases, the length of the row and column lines increase, thus increasing t_{row} and t_{col} as well. The row and the column time increases are muted because of decreases in geometry. Nevertheless, the random-access cycle time is approximately that given above for the basic DRAM. Thus, DRAM designers adopted an organization similar to that of the wide-word memory discussed in the above subsection and shown in Figure 4.27.

The page-mode DRAM is shown in Figure 4.31. With this organization, B arrays (replication of the basic DRAM of Figure 4.28) are accessed simultaneously with the row address. In parallel, one bit from each bit plane is transferred to the I/O pins; the approximate bandwidth is shown in the figure. Because the arrays are stacked, each array can be smaller and the random access time is reduced over that of a single array

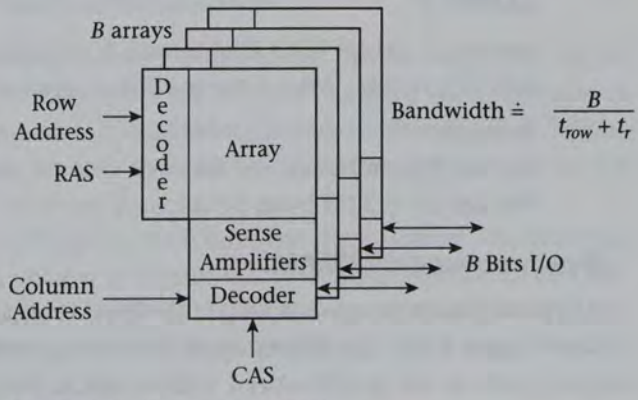


Figure 4.31 Page-mode DRAM

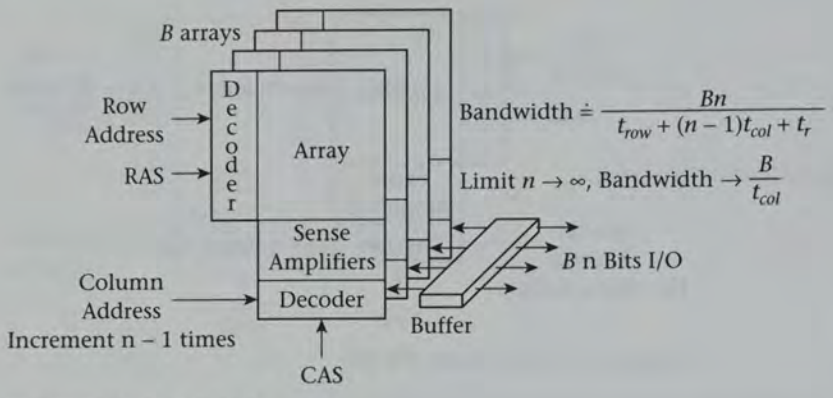


Figure 4.32 EDO DRAM

of equal capacity. If B is 4, the bandwidth of this organization is approximately twice that of the nibble-mode DRAM.

Page-mode DRAMs became practical with the maturity of VLSI and the availability of high pin packages. For example, a 1M bit × 16 requires 20 address pins and 16 I/O pins plus various control pins.

EDO DRAM. The extended data out DRAM, also know as hyperpage, adds an output buffer to the page-mode DRAM, as shown in Figure 4.32, and uses the incrementing feature of the nibble-mode DRAM. The use of this device is to perform a burst transfer of n words of B bits. The first word of B bits read from the arrays is stored in the buffer; this operation takes a full access time of t_{row}. Then the column addresses are incremented to read the next B bits while the contents of the buffer are transferred to the I/O pins. For large values of n, the streaming bandwidth approaches B/t_{col}.

When compared with that of the page-mode DRAM, the random access time ($n = 1$) has not been improved with this organization but the streaming rate is substantially improved. This improvement is important for applications such as image processing and loading caches. Clearly, the maximum value of n is the number of columns of a single array. If the addresses wrap around to another row, a new row access is required.

EXAMPLE

Assume a 4M-bit EDO DRAM with 8 I/O pins. The array has a column access rate of 33 MHz. What is the maximum streaming bandwidth of this device?

Solution

As one byte is transferred for each column cycle, the streaming bandwidth of this device is 33M bytes per second.

Synchronous-Link DRAM

A typical synchronous-link DRAM (SLDRAM) organization is similar to that of the EDO of Figure 4.32. The difference is that the system provides a clock to the DRAM and all controls are synchronized to that clock. The data input and output are also

Petitioner Exhibit 1053, Page 182 of 334

synchronous to the system clock. This organization permits a more robust memory system design as maximum and minimum access times are avoided. The SLDRAM is specified by an IEEE standards working group; the details of the SLDRAM organization are given by Gillingham and Vogley (1997).

The core of the synchronous DRAM is still the asynchronous device of Figure 4.28. The major difference is that most of the timing control is now on the DRAM chip. Multiple timing signals do not need to be generated and propagated on the memory board.

EXAMPLE

Assume the following DRAM parameters: $t_{row} = 70$ ns, $t_{col} = 25$ ns, and $t_r = 5$ ns. What is the bandwidth (BW) in bits/nanoseconds for the basic, nibble mode, page mode with four arrays, and EDO DRAMs with four arrays and $n = 8$?

Solution

$$\text{basic BW} = \frac{1}{70+5} = \frac{1}{75} = 0.013 \text{ bits/ns},$$

$$\text{nibble BW} = \frac{4}{70+(3\times 25)+5} = \frac{4}{150} = 0.026 \text{ bits/ns},$$

$$\text{page BW} = \frac{4}{70+5} = \frac{4}{75} = 0.053 \text{ bits/ns},$$

$$\text{EDO BW} = \frac{4\times 8}{70+(7\times 25)+5} = \frac{32}{250} = 0.128 \text{ bits/ns}.$$

Comment

Each step upward in the complexity of organization approximately doubles the bandwidth of the DRAM. However, the page and the EDO have three additional I/O pins. Note that the burst mode bandwidth limit for the EDO is 0.16 bits/ns for very large values of n.

4.5 VIRTUAL- AND REAL-ADDRESS CACHES

Figure 4.11 shows that the address presented to the cache can be either the virtual address from the processor or the translated real address. Both of these techniques have been used with modern processors. The advantages and disadvantages of the two approaches are presented in Table 4.13. As indicated in this table, popular processors use a form of real-address cache addressing.

Real-address caches can be pipelined to overcome the performance disadvantage of this design. With this design, shown in Figure 4.33, the cache set index and displacement are provided from the virtual-address displacement, which is untranslated. While the cache and its tags are being accessed, the page name is being translated and then compared with the real tags from the cache.

Petitioner Exhibit 1053, Page 183 of 334

TABLE 4.13 VIRTUAL- AND REAL-ADDRESS CACHES

Design	Advantages	Disadvantages
Virtual Address: Used on: Intel i860 SPUR	No added latency for address translation time. No limit on the size of the cache (can use as many bits of the virtual address as desired).	If only a portion of the virtual address is used for the cache address (to reduce the number of tag bits), different virtual addresses can point to the same cache address, called a synonym or alias.
Real Address: Used on: VAX 11/780 Pentium Pro* IBM RS/6000* IBM S370	Does not have the synonym or alias problem.	Latency added because of the need to translate the address. Latency can be small with a small dedicated TLB.

*Pipelined

Pipelining is described in detail in Chapter 6. However, it should be clear from Figure 4.33 that as long as there is a steady stream of valid addresses to this cache, there will be one cache access per clock. The stream of addresses is interrupted when there is a cache miss or some other delay.

Note that because of the limited number of virtual address displacement bits, 10–14, the size of the cache may be limited. Figure 4.33 shows a direct cache; however, *n*-way set associative caches can also be used to increase the size of the cache. For a pipelined real-address cache, the maximum cache size is

page size × degree of set associativity.

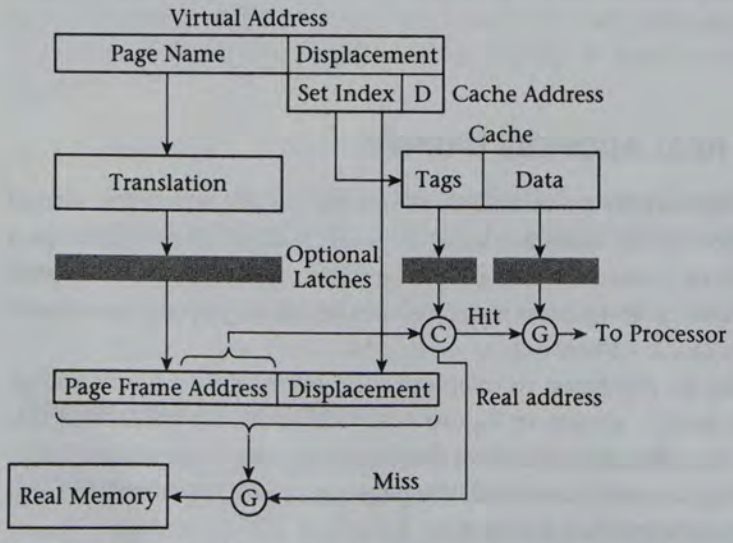


Figure 4.33 Pipelined real-address cache (courtesy Jones and Bartlett Publishers)

Petitioner Exhibit 1053, Page 184 of 334

EXAMPLE

A byte-addressable processor has a 2K-byte page and a 16-byte block. What degree of set associativity is required for providing a cache of 8K bytes with a pipelined real address cache?

Solution

The virtual address has an 11-bit displacement field. 4 bits are taken from the virtual displacement for the cache displacement into the 16-byte block. This leaves 7 bits for the set index. The required degree of set associativity is

$$\frac{\text{desired cache size}}{\text{page size}} = \frac{2^{13}}{2^{11}} = 4.$$

Comment

This design is identical to the Intel i486 on-chip instruction and data caches. A student may wonder why the number of sets does not enter into the determination of the maximum cache size.

4.6 SEGMENTED VIRTUAL MEMORY

Most of the early thinking on virtual memory systems was based on the idea that real memory was very expensive. For example, 1M byte of core memory for an IBM S360 sold for $1.3M in 1965! Because of the cost of memory, the idea of using fixed-length pages was considered wasteful by many designers because, on average, 1/2 page of real-memory space would be wasted for each allocated working set.[49] It was posited that if the allocation unit is a single AU rather than a page, the exact amount of real memory needed could be allocated for each working set.

This line of thinking led to the concept of a segment, which is a variable-length page with granularity of one AU. Thus on a segment fault, a segment would be loaded into real memory from the virtual-memory disk. However, segmentation leads to a lot of problems. For example, if the system wants to load a segment of 3456 bytes, a vacant contiguous portion of real memory must be found or created that is equal to or larger than 3456 bytes. Unlike paged systems, in which a simple bit vector is used to indicate the use of a page frame, a segmentation system must identify the vacant portions of memory by starting address and length. This information is usually stored in a list that must be searched to find vacant contiguous locations, as the segment cannot be subdivided.

Free space (space that is not allocated to active segments) in real memory must be managed and is usually compacted into one large address space. This means that from time to time the operating system must move the allocated segments in a compact space, reclaim the deallocated space into free space, and compact the free space into contiguous locations. This overhead process has been given the quaint name of compacting garbage collection.

One of the characteristics of a segmentation system is that segments can overlap for sharing of programs and data. Two or more segments can contain the same AUs.

[49] Working set is the set of pages allocated to a program that ensures good performance.

Petitioner Exhibit 1053, Page 185 of 334

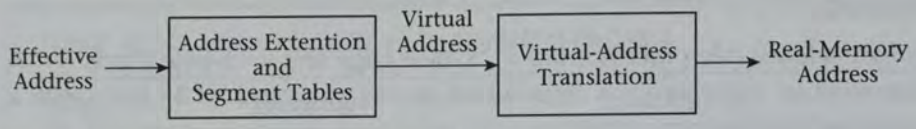


Figure 4.34 Paged segmented address translation

The benefit of sharing opens the possibility of programs that will have bugs; the responsibility to prevent these bugs is given to the software.

After a great deal of research and engineering work was expended on making pure segmentation systems work, the idea was abandoned. There was just too much hardware and software overhead, and the price of memory, the primary justification for segmentation, was going down. However, segment systems are used today in two forms: paged segmentation and virtual segmentation.

Paged Segmentation

Paged segmentation systems are used in most computer systems today for program management purposes. Instead of the allocation unit of one AU for the pure segment system, paged segmentation allocates at the page level. The management of the segments (a collection of pages) is done by the operating system, with some hardware support, and the hardware and software manage pages as described in Section 4.2.

With a paged segmented system, the effective address is extended by concatenation of an extension held in a control register or a table indexed by a portion of the effective address bits, shown in Figure 4.34. Details of the virtual-address translation tables are shown in Figures 4.6 and 4.8.

IBM S370

The complete segmentation and paging tables for the IBM S370 virtual-memory system are shown in Figure 4.35. The 24-bit logical address (effective address) is extended by 18 bits from control register 1 to give a 42-bit virtual address. The S370 24-bit logical address is divided into a segment index, page index, and byte index or displacement.

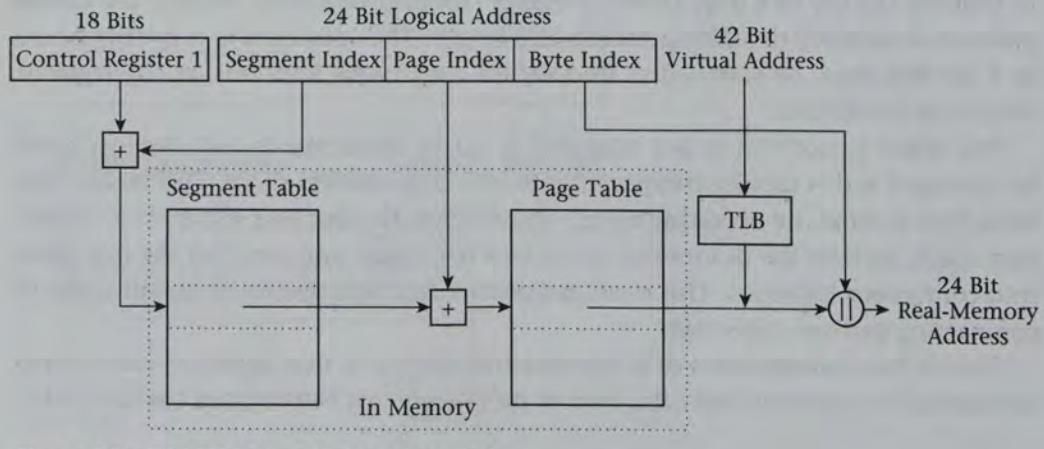


Figure 4.35 IBM S370 virtual-address translation

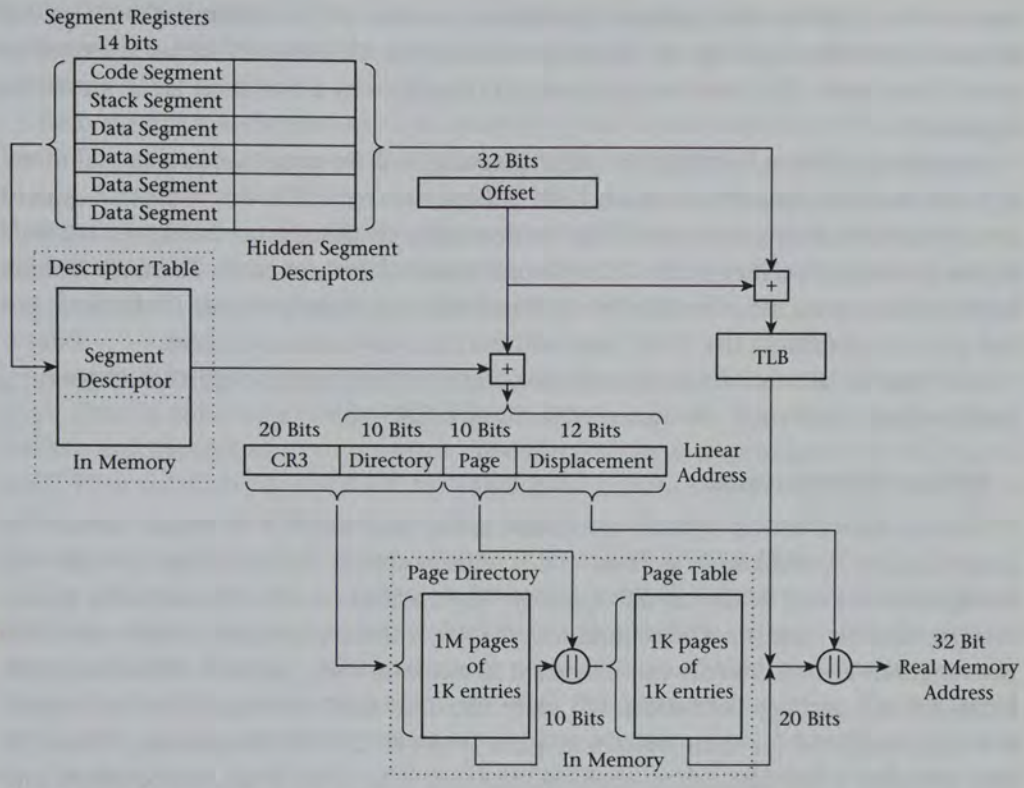


Figure 4.36 Pentium Pro segmentation

The length of the fields in the logical address vary since four configurations can be invoked. The configurations are

64K-byte segment, 4K-byte page 1M-byte segment, 4K-byte page
64K-byte segment, 2K-byte page 1M-byte segment, 2K-byte page

The control register bits are added to the segment index to index into the segment table. Bits from the segment table entry are added to the page index to index into the page table. Bits from the page table are concatenated with the byte index to give the 24-bit real-memory address. The specific allocations to control register 1, the segment table entries, and the page table entries are discussed in Section 8.2.

Because the segment and the page tables are in real memory, these tables are bypassed by the TLB holding pretranslated information.

Intel Pentium Pro

The segmentation scheme of the Intel Pentium Pro is shown in Figure 4.36. The descriptor table has the function of establishing a base address for the linear address. The length of the segment is set by an entry in the segment descriptor table.

14 bits of a 16-bit segment selector, found in a hardware register, indexes into the descriptor table that is allocated to real memory. There are six segment selector registers: one for a code segment, one for a stack segments, and four for data segments.

Petitioner Exhibit 1053, Page 187 of 334

One of the fields of the segment descriptor provides a base address that is added to the 32-bit offset, giving the 32-bit linear address. The linear address is extended with 20 bits from CR3 and the page name is translated by a two-level table, shown in Figure 4.6.

Because the descriptor table, the page directory, and the page table are in real memory, the memory-based translation path is bypassed by a TLB and hidden segment descriptors in hardware registers. The hidden segment descriptors hold pretranslated segment descriptors, identical to those found in the descriptor table. There is a hidden segment descriptor associated with each of the six segment selectors. Thus there is a fast path for obtaining the 20-bit base address for a real-memory access.

The specific allocations to the selector, segment descriptor, page directory entry, and the page table entry are discussed in Section 8.2.

Virtual Segmentation

Virtual segmentation systems are found today in a number of highly interactive programming systems such as Lotus 1-2-3. This system is highly dynamic, with cells being allocated and deallocated frequently. The number of cells allocated and deallocated is variable, and a paged segmentation system is too inflexible. Thus a primitive pure segmentation system is implemented to automatically perform allocation, deallocation, and garbage collection. On early PCs that used i286 and i386 processors, the time overhead for these tasks was quite noticeable. MSDOS versions of this system even had a red signal that would tell the user when allocation, deallocation, and garbage collection were occurring.

4.7 DISK MEMORY

The lowest-level memory in the hierarchy is the disk. By inference we mean the hard disk in PC terminology; other forms of disks, such as floppies, are not permanently connected and can be considered to be archival. This distinction is discussed further in Chapter 7.

Disks are organized in a series of layers, as shown in Figure 4.37. A disk consists of a number of platters, and each platter has one or two surfaces. Each surface has a number of tracks that are divided into sectors consisting of an even power of 2 bytes. For some disk designs, each track has the same number of sectors. Thus the bit density decreases for the tracks near the periphery of the disk. For other designs, the bit density per track is constant and the number of sectors per track increases near

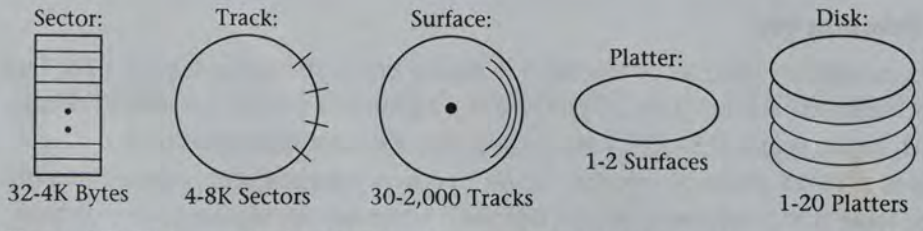


Figure 4.37 Disk organization (courtesy Jones and Bartlett Publishers)

Petitioner Exhibit 1053, Page 188 of 334

the periphery of the disk. This technique is called a zoned disk. Usually the sector is the addressable unit of the disk; the disk sector should not be confused with the sector of a cache.

Each surface usually has one arm holding a read and a write head or a read/write head. The write heads convert the digital signal to a magnetic signal that polarizes the magnetic surface into a string of 1s and 0s. The read head, because of the rotation of the disk, senses the magnetic flux change and generates a string of ones and zeros. Note that the bit areas on the disk are not patterned (as with DRAM memory chips) but depend on the air gap in the write head to delineate the bit areas as the platter rotates.

The write or read transport time of a disk is the sum of a number of time elements. First, there is the rotational latency. Many disks have only one read/write head per surface and the disk must rotate until the desired sector is in position to be written or read. With rotation rates of 3600 to 10000 RPM, the maximum rotation latency is a full revolution time of 16.6 to 6 ms. The minimum rotation latency is zero, and the average latency is one half of a revolution or, for 5200 RPM, 5.75 ms.[50]

The second component of latency is seek latency. The head must be positioned over the addressed track adding to the latency. Positioning requires that the arm be moved, starting from an at-rest position, accelerated, and then positioned over the desired track. The minimum latency is zero if the head is already on the addressed track, but the maximum latency can be approximately 20 ms or more, depending on the number of tracks on a surface. There are various techniques for reducing seek latency for multiprogrammed computers.

There is additional latency from the operating system of approximately 1 to 2 ms, depending on the complexity of the page fault handler and the speed of the processor. The operating system or page fault handler must translate the binary virtual address into the physical addresses of the disk system: disk, platter, surface, track, and sector.

In addition to the disk latency, the data transfer time to and from the disk must be included in the transport time. The time to actually read or write the data is determined by the size of the sector, the number of contiguous sectors to be transferred, and the bit rate of the read/write process. With modern disks, the transfer rate is in the range of 5M to 15M bits/s for a single head. When multiple heads are used to transfer parallel bit streams, the total transfer rate is a multiple of the single-head rate.

The total disk transport time for a read or write has three latency components plus the transfer time:

$$\begin{aligned}\text{transport time} = {} & \text{rotational latency} + \text{seek latency} \\ & + \text{operating system latency} + \text{transfer time.}\end{aligned}$$

EXAMPLES

1. A disk rotates at 5200 RPM and the data transfer rate is 5M bytes/s when driven from 60 cycle power. The average seek time is 25 ms, and the sector is 1K bytes.

[50] Laptop computers that stop the disk rotation to conserve battery power will have even longer latencies because of the time (5– 20 s) required for starting the disk.

Petitioner Exhibit 1053, Page 189 of 334

A 100-MIPS computer has a 4K-byte page and the page fault handler requires the execution of 750,000 instructions.

a. What is the transport time for a page with 60 cycle power?

b. What is the transport time for a page with 50 cycle power?

Solution

a. The latency is $[(5.75\times10^{-3})+(25\times10^{-3})+(7.5\times10^{5})]/100\times10^{6}\times10^{-3} = 38.25$ ms. The transfer time is $(4\times1024)/(5\times10^{6}) = 0.819$ ms. The transport time is $(38.25\times10^{-3})+(0.819\times10^{-3}) = 39.07$ ms.

b. At 50 cycles the average rotational latency is 6.9 milliseconds and the data transfer rate is 4.16M bytes/s. The latency is $(6.9\times10^{-3})+(25.00\times10^{-3})+(7.5\times10^{-3}) = 39.49$ ms. The transfer time is $(0.819\times10^{-3})\times60/50 = (0.819\times10^{-3})\times1.2 = 0.983$ ms. The transport time is $(39.49\times10^{-3})+(0.983\times10^{-3}) = 40.47$ ms.

2. A disk has 128 sectors per track with 512 bytes per sector. The rotation rate is 5200 RPM. What is the transfer rate in bytes/second?

Solution

The rotation time is $60/5200 = 11.5$ ms. The transfer rate is $(128\times512/11.5)\times10^{-3} = 5.7$M bytes/s.

Disk Caches

The example above for determining the transport time of a disk illustrates one of the problems with a memory hierarchy. Even though the page fault rate is very low, when a page fault does occur there is a major disruption of program execution while the page fault is being resolved. One solution to this problem is the use of a cache between the real memory and the disk, as shown in Figure 4.38.

There are two options for implementing a disk cache: (1) implement a hardware cache or (2) implement a virtual disk cache in real memory.

On the surface, the first option is straightforward: a designer need only decide on the size and select an organization. A major problem is to decide on the organization of the cache tags. The virtual address produced by the processor is a binary number but the disk address is nonbinary: disk, platter, track, and sector. If these fields are to be allocated to the cache tags, provisions must be made for disk size expansion or for replacement disks that are organized differently. The solution to this problem is to use virtual address in the tags and map the virtual address to the disk address in software when there is a disk cache miss.

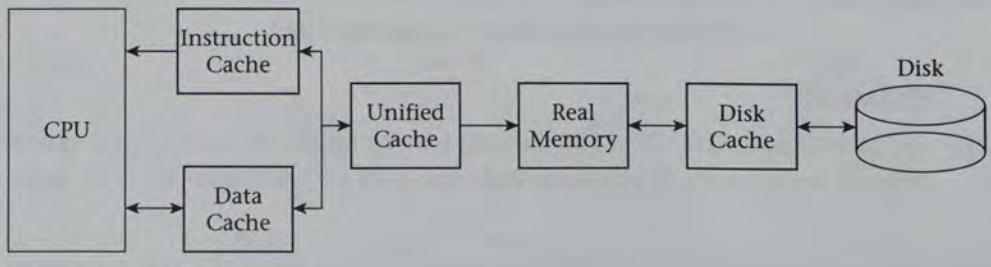


Figure 4.38 Disk cache

Petitioner Exhibit 1053, Page 190 of 334

The second solution to the disk cache design problem allocates space in real memory for a virtual disk cache and performs all the required functions in software. Even if these functions take several hundred thousand instructions, with a 100-MIPS processor the time is small compared with the transport time of a disk-microseconds rather than milliseconds.

The MSDOS operating system provides for a virtual disk cache. Depending on the amount of memory in the system, the disk cache– a program called SMARTDrive – can be as large as 2M bytes. The page size can be selected by the user, an 8K-byte page being typical. A user also sets the size of the cache, and hit rates of 0.7 to 0.8 are achievable. SMARTDrive supports reads and writes to/from only the file system; it does not support virtual memory.

Windows95 has two virtual caches: one for the virtual-memory system consisting of a program called VCACHE, and the other for CD ROMS, called CDFS. Unlike the SMARTDrive of MSDOS, the size of both of these caches is dynamically set by the operating system. The cache sizes are set by the software to be as large as possible depending on the amount of free memory at any time. The page size of the VCACHE is fixed at 4K bytes.

EXAMPLE

A memory system with a transport time of 50 ms for a 4K-byte page is upgraded with a disk cache. The $P_{hit} = 0.75$ and the transport time for a transfer of a page from the disk cache to real memory is 50 μs. What is the speedup in the transport time given by the disk cache?

Solution

$$\begin{aligned}\text{Weighted mean transport time} &= (0.75 \times 50 \times 10^{-6}) + (0.25 \times 50 \times 10^{-3}) \\ &= 12.5 \times 10^{-3}\ \text{s},\end{aligned}$$

$$\text{speedup} = \frac{50 \times 10^{-3}}{12.5 \times 10^{-3}} = 4$$

Comment

This is a good example of Amdahl's law. Three fourths of the time was speeded up almost an infinite amount (RAM speed versus disk speed) and one fourth of the time was not speeded up. Therefore the overall speedup is simply determined as

$$\text{speedup} = \frac{1}{1-\alpha} = \frac{1}{1-0.75} = \frac{1}{0.25} = 4.$$

4.8 SYSTEM OPERATION

We have looked at the components of a virtual-memory system with caches. Now we can tie all of these components together and see what happens to an address and data as they make their way through the system. For expository purposes we consider a

Petitioner Exhibit 1053, Page 191 of 334

data read operation through a system with the following components.

- L1 data cache
  - Write hit: write through
  - Write miss: no-write allocate
- L2 level unified cache
  - Write hit: write through
  - Write miss: no-write allocate
- Interleaved real memory: write through (RM)
- Translation lookaside buffer
- Inverted page table
- Disk cache (DC)
  - Write hit: write back
  - Write miss: no-write allocate
- Disk

A preliminary event table is shown in Table 4.14. The table is called preliminary because each one of the actions may require its own event table. Because the cache policies are write through, the caches, real memory, and the disk cache are assumed to be coherent.

Because the cache policies are write through, the caches can be overwritten when a block is moved to a higher level. With some of the events, the actions can take place in parallel reducing the time for the event. Designers search for parallel paths that can be used to reduce the time of an event.

TABLE 4.14 READ ACCESS MEMORY SYSTEM EVENTS

L1	L2	RM	DC	Disk	Action
Hit	×	×	×	×	Read datum from L1 into processor
Miss	Hit	×	×	×	Read datum from L2 into processor Overwrite block from L2 into L1
Miss	Miss	Hit	×	×	Translate address in TLB or tables Read datum from RM into processor Overwrite block from real memory into L2 Overwrite block from real memory into L1
Miss	Miss	Miss	Hit	×	Translate address into disk cache addresses Read datum from disk cache into processor Overwrite page into RM Overwrite block into L2 Overwrite block into L1 Update TLB
Miss	Miss	Miss	Miss	Hit	If page in disk cache is modified, evict to disk and read page into disk cache. If page in RM is modified, evict to disk and read page into RM. Restart the read access

For example, the case of L1 and L2 cache misses and RM hit; as soon as the datum is in the processor, the processor may proceed. Also, the transfers into the two caches can take place in parallel with the transfer to the processor. However, if there is a following cache access before the transfers are complete, that access may stall. Another example is the TLB, which can be searched in parallel with the start of the table search for address translation.

REFERENCES

Cragon, H. G. (1980). "The Elements of Single-Chip Microcomputer Architecture," *Computer*, **13**:10, pp. 27– 41.

Flynn, M. J. (1966). "Very High-Speed Computing Systems," *Proceedings of the IEEE*, **54**:12.

Gee, J. D., Hill, M. D., Pnevmatikatos, D. N., and Smith, A. J. (1993). "Cache Performance of the SPEC92 Benchmark Suit," *IEEE Micro*, **13**:4, pp. 17–27.

Gillingham, P. and Vogley, B. (1997). "SLDRAM: High-Performance, Open-Standard Memory," *IEEE Micro*, **17**:6, pp. 29– 39.

IBM (1976). "IBM System/370 Principles of Operation," Order GA22-7000-5.

Kilburn, T., Edwards, D. B. G., Lanigan, M. J., and Summer, F. H. (1962). "One-Level Storage System," *IRE Transactions on Electronic Computers*, **7**:4.

Lewis, T. G. and Cook, C. R. (1988). "Hashing for Dynamic and Static Internal Tables," *Computer*, **21**:10, pp. 45–56.

Milligan, M. K. and Cragon, H. G. (1995). "Processor Implementations Using Queues," *IEEE Micro*, **15**:4.

Saavedra, R. H. and Smith A. J. (1993). "Measuring Cache and TLB Performance and Their Effect on Benchmark Run Times," Report USCB/CSD 93/767, University of California, Berkeley.

Smith, A. J. (1987). "Line (Block) Size Choice for CPU Cache Memories," *IEEE Transactions on Computers*, **C-36**:9.

EXERCISES

4.1 There are two write hit and two write miss policies, giving four possible designs. List these four designs and discuss the feasibility of each.

4.2 An instruction cache is large enough to hold a loop. What is the instruction cache P_{miss} for a loop that is executed 50,100, and 200 times?

4.3 Derive a model for the effective cache latency of Figure 4.24. Take the first derivative, set to zero, and solve for the minimum.

4.4 Comment on the performance of an interleaved memory if the degree of interleaving is a prime number: 3, 5, 7 etc., rather than an even power of 2.

4.5 Discuss the problem of synonyms or aliases that are characteristics of virtual-address caches.

4.6 The issue of increasing the size of a pipelined real-address cache has been discussed. Using the data on the data miss rate for the SPEC benchmarks, determine the change in P_{miss} of the i486 cache for the degree of associativity: 1, 2, 4, and 8.

4.7 Refer to the discussion on zoned disks on disks in Section 4.7. Discuss the issue of transfer rates for the two techniques of assigning sectors to tracks.

Petitioner Exhibit 1053, Page 193 of 334

4.8 Discuss the problems associated with a branch-and-link instruction that places the link in memory if the memory system has a cache.

4.9 Modify the t_{elrm} model found on model derived in Table 4.4 to comprehend a TLB.

4.10 Explain why a valid bit is needed to detect a cache hit. Why isn't a tag comparison sufficient?

4.11 The figures showing direct and inverted page tables do not show valid bits. Are valid bits needed? Explain your answer.

4.12 Figure 4.26 shows the S370 address translation path. Assume the following. Accessing the segment table and the page table in real memory takes four memory cycles each. Accessing the TLB takes 0.1 memory cycles. The TLB has a hit rate of 0.95. What is the effective address translation time for the S370?

4.13 Figure 4.33 shows that the Intel Pentium Pro TLB uses as its input the virtual or the linear address. How does this architecture overcome the delay of accessing the descriptor table?

4.14 Explain how the Intel page directory table is indexed with 30 bits if the real memory address is 32 bits.

4.15 Rework the example on the weighted average number of probes for the inverted student class file on file of Table 4.8 and Figure 4.8. Use two hashing schemes: the hash is the three least-significant digits and the hash is the one least-significant digit. Recall that, when the two least-significant digits are used, the weighted average number of probes is 1.32.

4.16 Compute and plot the effective cache latency for bus widths of 1, 2, 4, 8, 16, and 32 bytes by using the parameters of the example on page 167. Use a 64 byte block.

4.17 Table 4.8 shows P_{miss} for instruction caches and Table 4.9 shows P_{miss} for data caches. Is it possible to construct a reasonable estimate of the data in Table 4.10 by using the data from Tables 4.8 and 4.9? If so, construct the table; if not, explain why not.

4.18 The text mentions that a five-way set associative cache can be designed. Explain the advantages and the disadvantages of this design. Is a prime-number-way interleaving advantageous or disadvantageous?

4.19 Construct the event table, similar to Table 4.13, for a data write.

4.20 Assign miss rates and action times for Table 4.13 and compute a weighted average memory write time.

4.21 Redraw Figures 4.12 and 4.13 to reflect the modern cache designs that have one block slot per sector.

Petitioner Exhibit 1053, Page 194 of 334

FIVE

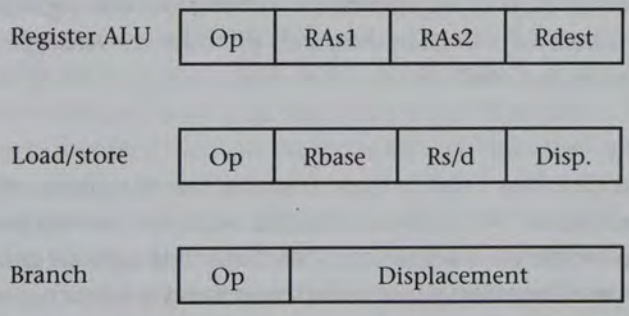
PROCESSOR CONTROL DESIGN

5.0 INTRODUCTION

Recall that Figure 3.1 shows the interface between software translation and hardware interpretation of a program written in a high-level language. This chapter now describes two methods of performing the hardware translation: hardwired control and microprogramming. Today, a state diagram is routinely used to show the sequence of events required for interpreting an instruction.

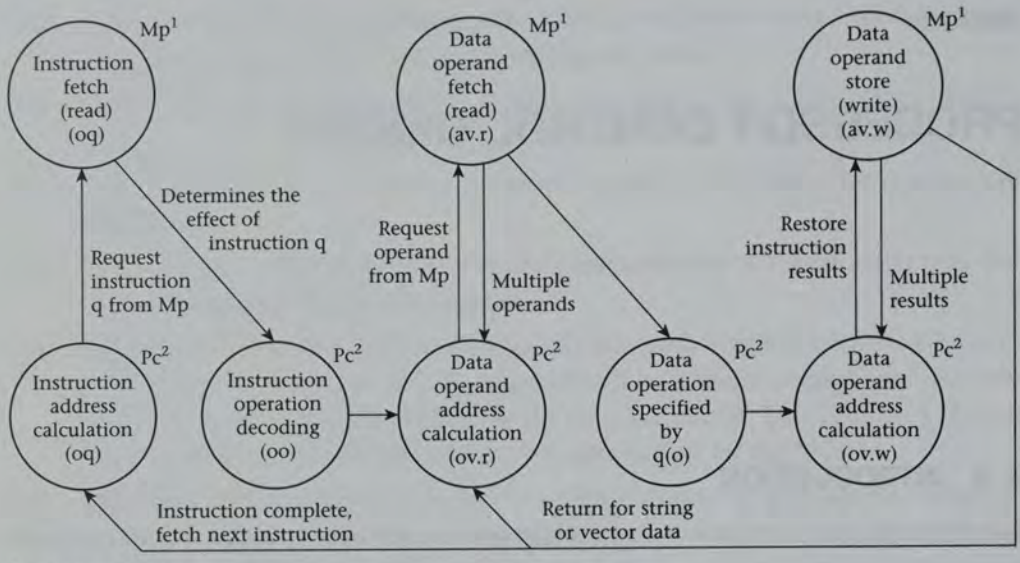
A state diagram for instruction interpretation is shown in Figure 5.1 (Bell and Newell 1971). This state diagram represents a very early ISA and shows the serial execution model of the interpretation cycle; one instruction completes before another instruction begins. The three upper states are controlled by the memory, and the five lower states are controlled by the processor. Also, any state can be null to represent the interpretation of another instruction.

As a vehicle for describing hardwired and microprogrammed control, a simple load/store register file machine is used as an example. The example processor has three instruction classes: register ALU, load/store, and branch. The instruction formats for the three classes of instructions are shown below:



The register ALU instructions have two source and one destination operands, all in the register file. The addressing mode for the load/store instructions is base plus displacement with the base value stored in the register file. The branch instruction tests the sign bit of R1 much as the von Neumann ISA tested the contents of the accumulator. The branch target address is the program counter plus the displacement.

Petitioner Exhibit 1053, Page 195 of 334



[1]Mp controlled state
[2]Pc controlled state
Note: Any state may be null

State name	Time in a state	Meaning
soq/oq	toq	Operation to determine the instruction q
saq/aq	taq	Access (to Mp) for the instruction q
so.o/o.o	to.o	Operation to decode the operation of q
sov.r/ov.r	tov.r	Operation to determine the variable address v
sav.r/av.r	tav.r	Access (to Mp) read the variable v
so/o	to	Operation specified in q
sov.w/ov.w	tov.w	Operation to determine the variable address v
sav.w/av.w	tav.w	Access (to Mp) to write variable v

Figure 5.1 Processor state diagram (courtesy McGraw-Hill Companies Inc.)

The type of ALU instructions and the field lengths of the instructions are ignored so that the ideas behind the design of the control unit are not obscured.

Before the details of the control are discussed, we must first look at what is controlled. For modern processor implementations the processor registers are configured into a data path. The data path to be described provides the resources to interpret the three instruction classes noted above.

Data Path

As discussed in Chapters 1 and 3, there are a number of registers within the processor. These registers are of two types: architected registers that can be accessed by the program (the register file for example) and implemented registers that are used by the control hardware as instruction side effects (the memory-address register for example). The control state machine must have the ability to load and store information in both register types.

In early computers, all of the processor registers were interconnected by point-to-point paths required by specific instructions. Modern design techniques, brought about by the desire for regular interconnections, connect these registers by means of one or more buses. A simple example of a data path is shown in Figure 5.2; this data

Petitioner Exhibit 1053, Page 196 of 334

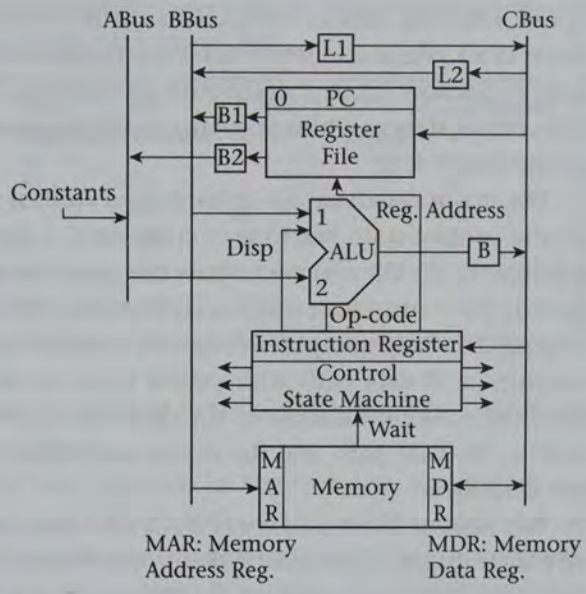


Figure 5.2 Data path

path will be used in the following discussions of hardwired and microprogrammed control design.

The data path has three buses so that dyadic operations can be performed in one clock. In addition to the three buses, there are some dedicated transfer paths that do not map well onto a bus. These dedicated paths are associated primarily with the instruction register (the register that holds the instruction while the instruction is being executed). The various fields of the instruction are directly connected to the needed destination. For example, the op-code is connected to the control state machine and to the ALU. The address displacement is a special input to the ALU. Register addresses are input to the register file.

Because the BBus and CBus need to communicate, two links (L1 and L2) are provided. Also note that the program counter is location 0 of the register file and that there is an input to the ABus for constants such as +1, 0, −1. A buffer is placed on the output of the ALU that is loaded with the results of each ALU operation. This buffer is needed because of the long logic path of the ALU. There is also a special path from memory to signal the control state machine when a wait is required. The control state machine provides sequencing through the states and provides the logic control signals that enable the transfers to/from the bus.

Low-performance data paths can be implemented with only one bus. A two-bus data path provided additional concurrency and additional performance. These one- and two-bus data paths are valid design options for situations for which cost is more important than performances, such as some embedded microprocessors.

5.1 HARDWIRED CONTROL

We first look at hardwired control. This is a technique whereby the designers commit the control design to a state machine. The advantage of this technique is primarily

Petitioner Exhibit 1053, Page 197 of 334

speed. One can usually design a hardwired control state machine that operates at a faster clock rate and in fewer states than a microprogammed controller, to be described later in Section 5.2. A characteristic of hardwired control is that the hardware cost of the control state machine is approximately proportional to the number of instructions in the macro ISA.

The major disadvantage of hardwired control is lack of flexibility. Once the design of the control state machine is completed, a change to correct a design error or an addition to the ISA may necessitate the complete redesign of the control state machine. And, if the control state machine is implemented on a chip, the complete chip must be redesigned. Early computer designers tended to make corrections to the control state machine and data path with yellow wire. As the quantity of yellow wire increased, one had a sure indication of the difficulty of this type of design. In the sections to follow, the data path and the design technique of hardwired control state machines are described.

The basis of hardwired control is a state machine that sequences through the states of a state diagram, generating the proper control signals to the data path at each state. The state diagram is the basis for the design of a hardware state machine. This design task is accomplished by five steps.

1. Construct the state diagram.
2. Identify transfer paths on the data path.
3. Make secondary state assignments.
4. Construct a state transition truth table.
5. Write the input equations for the flip flops.

Construct State Diagram

A load/store architecture has a simple state diagram, as shown in Figure 5.3. The first step in interpretation is to fetch an instruction pointed to by the program counter. The instruction is decoded, and if the instructions require register values, the registers are accessed. The next state either computes the effective address (EA) for a memory operation or executes the function called for by the op-code. The memory state performs its read/write operations for load/store instructions. The write to register file state writes the result of an operation or the datum obtained by a load instruction. Finally, the PC

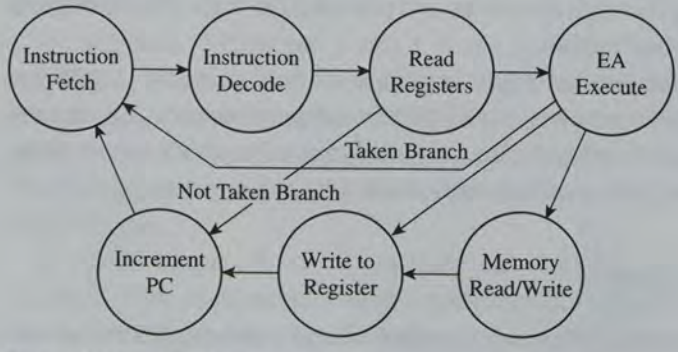


Figure 5.3 Processor state diagram

Petitioner Exhibit 1053, Page 198 of 334

TABLE 5.1 INSTRUCTION CLASSES AND STATE USE

	Inst. Fetch	Inst. Decode	Read Registers	EA Execute	Memory R/W	Write to Register	Increment PC
ALU	X	X	X	EX		X	X
Load	X	X		EA	R	X	X
Store	X	X	X	EA	W		X
T Branch	X	X	X	EA			
NT Branch	X	X	X				X

is incremented, and the process starts over. Table 5.1 shows the states of Figure 5.3 and the instruction classes that use those states.

The state diagram of Figure 5.3 is now redrawn in Figure 5.4 to show more details of the actual flow of instruction types through the state diagram. Note that all instructions use the instruction fetch and the instruction decode states. At the decoding state, the path diverges, depending on the instruction class. Because the instruction fetch and memory read/write will access memory, there may be wait states for a slow memory to respond. If the processor has caches, a wait is required when there is a cache miss.

Branch instructions, if taken, compute the target address and return to the instruction fetch state. If the branch is not taken, the program counter is incremented with a return to the instruction fetch state.

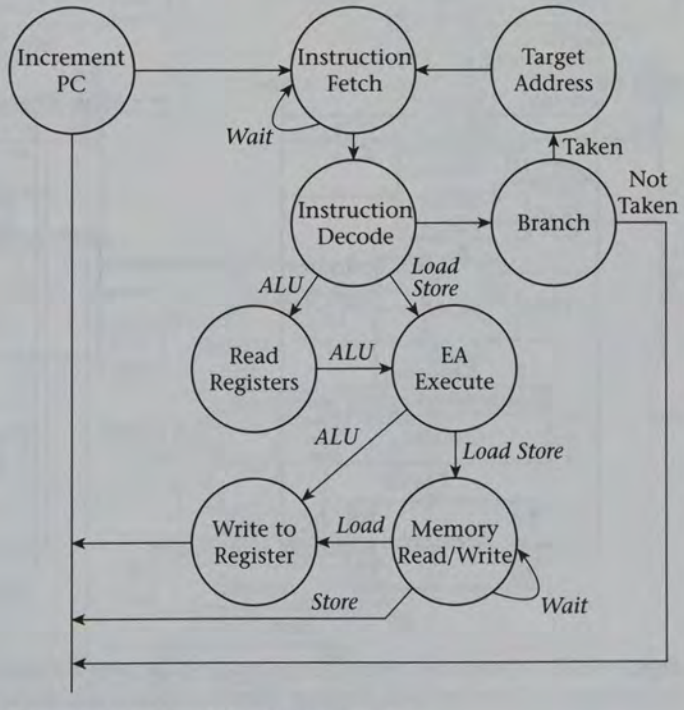


Figure 5.4 Interpretation state diagram

Identify Transfer Paths on the Data Path

The following tableau of nine drawings, Figures 5.5a–5.5i, show the flow paths that are enabled during each of the states in Figure 5.4. Transfers between registers take one clock. The memory cycle time is one clock plus any wait states. Some of the groupings of states are arbitrary; a transfer could have been placed with another function. The transfer paths are identified by shading and the number of clocks is listed for each of the nine functions.

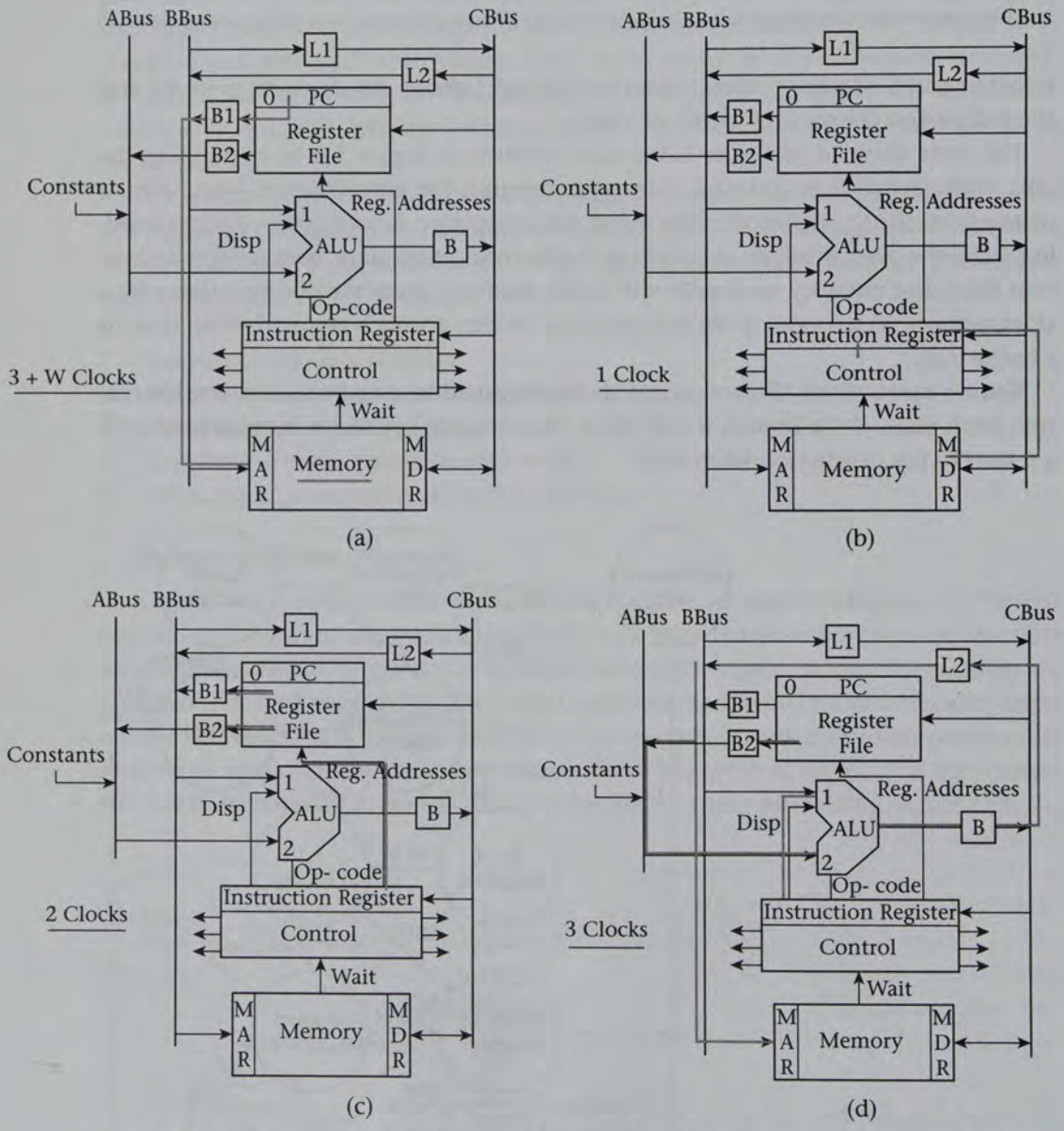


Figure 5.5 (a) Instruction fetch; (b) Instruction decode; (c) Register select; (d) Effective address; (e) Execute; (f) Memory read load; (g) Memory write store; (h) Memory increment PC; (i) Branch target address

Petitioner Exhibit 1053, Page 200 of 334

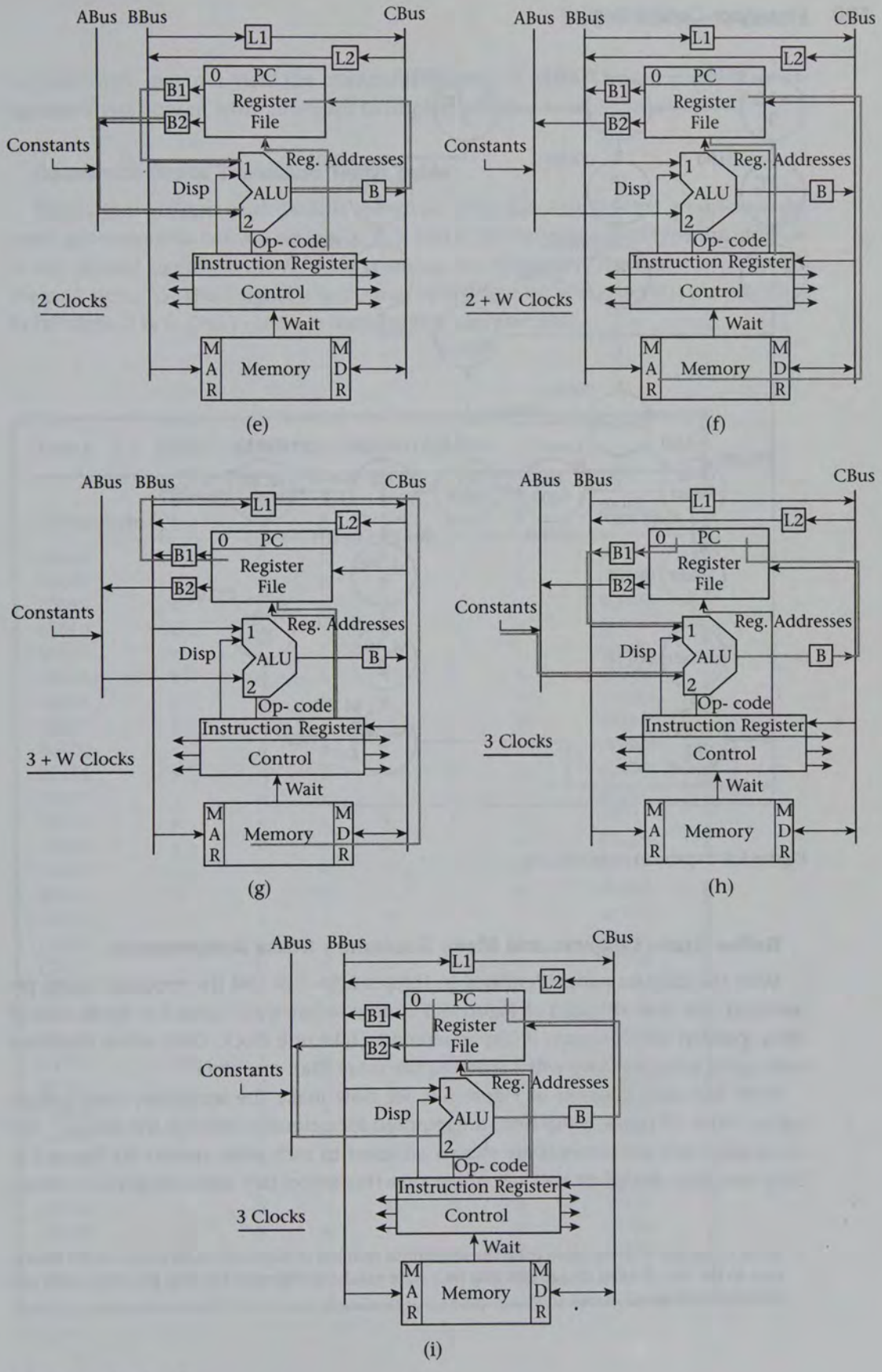


Figure 5.5 (*Continued*)

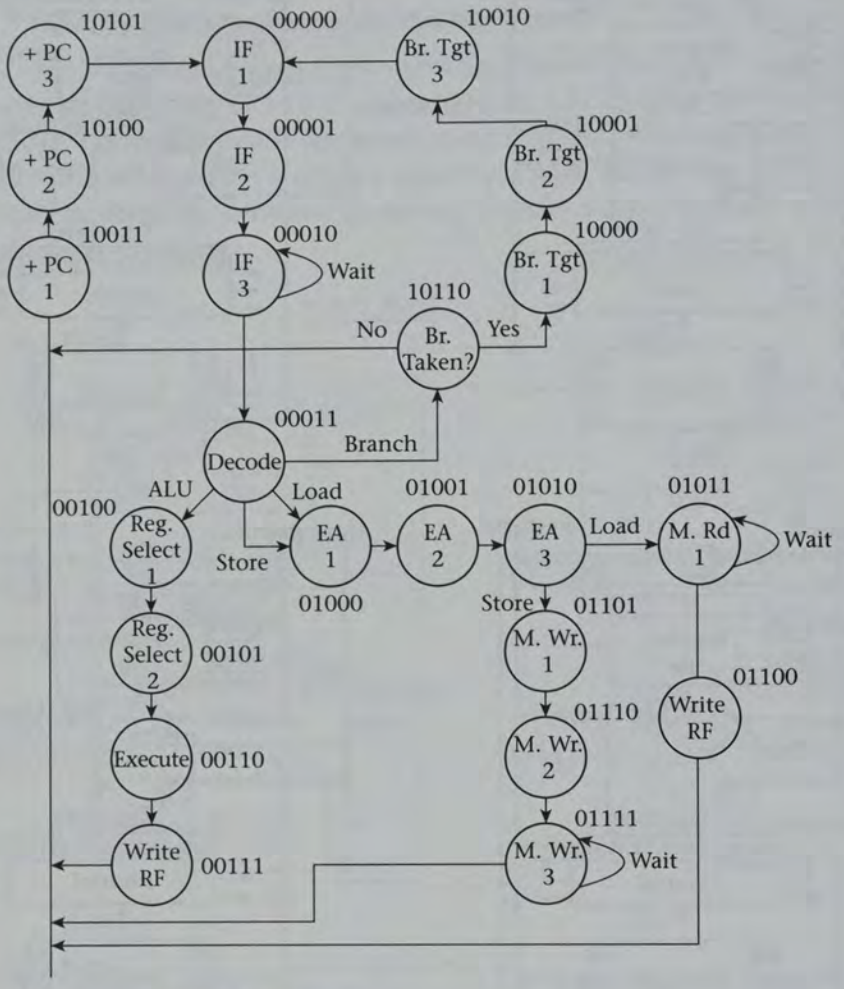


Figure 5.6 Expanded state diagram

Refine State Diagram and Make Secondary State Assignments

With the transfer paths identified in Figures 5.5a–5.5i and the required clocks per function, the state diagram of Figure 5.4 is now redrawn in Figure 5.6. Each state of the expanded state diagram is implemented to take one clock. Only when there is a wait signal from memory will a state require more than one clock.

With the state diagram of Figure 5.6, we now make the secondary state assignments. With 23 states, 5 flip flops are required for uniquely defining the states.[51] The secondary state assignments are shown adjacent to each state symbol in Figure 5.6. State machine designers will recognize that this secondary state assignment, when

[51] Some designers will use more than the minimum number of flip flops to be prepared for expansion in the ISA. A valid design can also be a state machine with one flip flop per state, only one of which will be a 1 at any time.

Petitioner Exhibit 1053, Page 202 of 334

implemented, may not yield the minimum number of gates. The secondary state assignment was chosen without regard to implementation complexity.

Construct State Transition Truth Table

The state transition truth table is shown in Table 5.2. Assume that an op-code decoder generates four control signals, A, L, S, and B, for the instruction types in addition to the control signals to the ALU for selecting the operation. The memory generates the wait signal W. The ? signal is generated by the sign bit of register R1; if a branch is to be taken, ? is 1. Don't cares are indicated by x in the table.

TABLE 5.2 STATE TRANSITION TRUTH TABLE

Present State	Branch ?	Wait W	ALU A	Load L	Store S	Branch B	Next State
00000	x	x	x	x	x	x	00001
00001	x	x	x	x	x	x	00010
00010	x	1	x	x	x	x	00010
00010	x	0	x	x	x	x	00011
00011	x	x	1	x	x	x	00100
00011	x	x	x	1	x	x	01000
00011	x	x	x	x	1	x	01000
00011	x	x	x	x	x	1	10110
00100	x	x	x	x	x	x	00101
00101	x	x	1	x	x	x	00110
00011	x	x	x	x	x	x	00111
00111	x	x	x	x	x	x	10011
01000	x	x	x	x	x	x	01001
01001	x	x	x	x	x	x	01010
01010	x	x	x	1	x	x	01011
01010	x	x	x	x	1	x	01101
01011	x	1	x	x	x	x	01011
01011	x	0	x	x	1	x	01100
01100	x	x	x	x	x	x	10011
01101	x	x	x	x	x	x	01110
01110	x	x	x	x	x	x	01111
01111	x	1	x	x	x	x	01111
01111	x	0	x	x	x	x	10011
10000	x	x	x	x	x	x	10001
10001	x	x	x	x	x	x	10010
10010	x	x	x	x	x	x	00000
10011	x	x	x	x	x	x	10100
10100	x	x	x	x	x	x	10101
10101	x	x	x	x	x	x	00000
10110	1	x	x	x	x	x	10000
10110	0	x	x	x	x	x	10011

Petitioner Exhibit 1053, Page 203 of 334

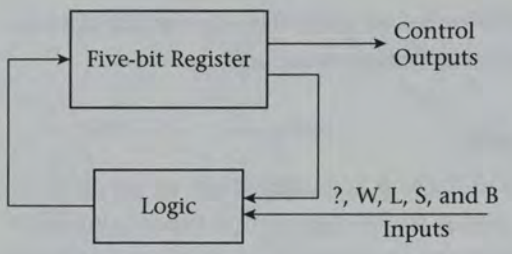


Figure 5.7 Control state machine

Write Input Equations for the Flip Flops

The control state machine, similar to that of Figure 1.1, is redrawn in Figure 5.7. The memory is a 5-bit register with inputs ?, W, A, L, S, and B along with the present state of the register bits. The control state machine transitions from state to state, generating control outputs that enable the transfer gates in the data path.

The 5-bit state register is implemented with SR flip flops named a, b, c, d, and e, reading from left to right in the secondary state assignment. The input equations for flip flop a are given below in minterm form as they would be implemented with a programmable logic array (PLA):

$$S_a = a'b'c'deB + a'b'cde + a'bcd'e' + a'bcdeW',$$
$$R_a = ab'c'de' + ab'cd'e.$$

EXAMPLE

There are 23 states shown in Figure 5.6 whereas the state transition truth table has 31 entries. Why the difference?

Solution

As designed, the state transition truth table must have one entry for every outarrow of the state diagram. Count them: there are 31.

We see that this implementation of the control state machine by using a data path results in the different classes of instructions taking a different number of clocks. The number of memory wait state clocks must be added to the number of clocks required for stepping through the expanded state diagram of Figure 5.6. Note that the logic paths of the branch instructions are the same for taken and not taken.

ALU:	11 + W clocks
Branch:	8 + W clocks
Load:	12 + 2W clocks
Store:	13 + 2W clocks

EXAMPLE

Assume that the weights of the instruction classes are ALU-0.30, branch-0.30, load-0.25, and store-0.15. Assume that the processor has a cache and that there

Petitioner Exhibit 1053, Page 204 of 334

are no wait states on a cache hit but 10 wait states on a miss. The P_{miss} of the cache is 0.10. What is the CPI of the processor?

Solution

The CPI of this processor is the weighted mean of the clocks for each class of instruction. The number of wait state clocks is determined by

$$\begin{aligned} W &= (1 - P_{miss}) \times 0 + P_{miss} \times 10 = 10 P_{miss} \\ &= 10 \times 0.1 = 1 \text{clock}. \\ CPI &= \text{ALU weight} \times \text{ALU clocks} + \text{branch weight} \times \text{branch clocks} \\ &\quad + \text{load weight} \times \text{load clocks} + \text{store weight} \times \text{store clocks} \\ &= 0.3(11 + 1) + 0.3(8 + 1) + 0.25(12 + 2) + 0.15(13 + 2) \\ &= 3.6 + 2.7 + 3.5 + 2.25 = 12.05. \end{aligned}$$

Comment

Computers with hardwired control and a data path similar to that of Figure 5.4 have CPIs in the range of 10 to 15. Thus this example is typical of computers of that type. Note that reducing P_{miss} by 1/2 would improve the CPI of this processor approximately 10%. Other improvements can come from eliminating clocks from the logic path. For example, one clock eliminated from the instruction fetch function will improve performance by approximately 8%.

5.2 MICROPROGRAMMED CONTROL

M. V. Wilkes first posited the idea that the best way to design a computer was to replace a hardwired control state machine with a simple control computer (Wilkes 1951). The control computer would have a very simple micro ISA and could be designed and built with few errors. The macro ISA would then be interpreted by a program, called a microprogram, running on the control computer having a micro ISA.[52] Figure 3.1 is redrawn in Figure 5.8 to show the two levels of software translation that are used with microprogrammed control.[53]

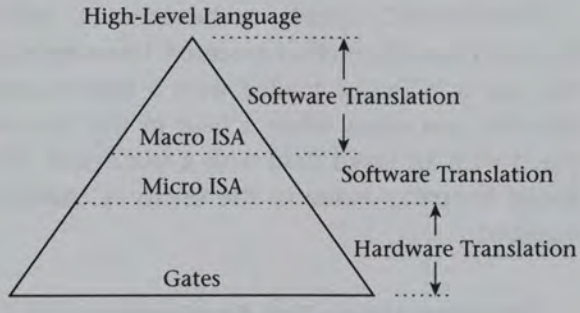


Figure 5.8 Microprogrammed lexical levels

[52] The terms macro ISA and micro ISA are used to distinguish between the two ISAs.

[53] The software that executes on the micro ISA is sometimes called firmware because it is stored in ROM and it is between the hardware and software; thus it is firm.

Petitioner Exhibit 1053, Page 205 of 334

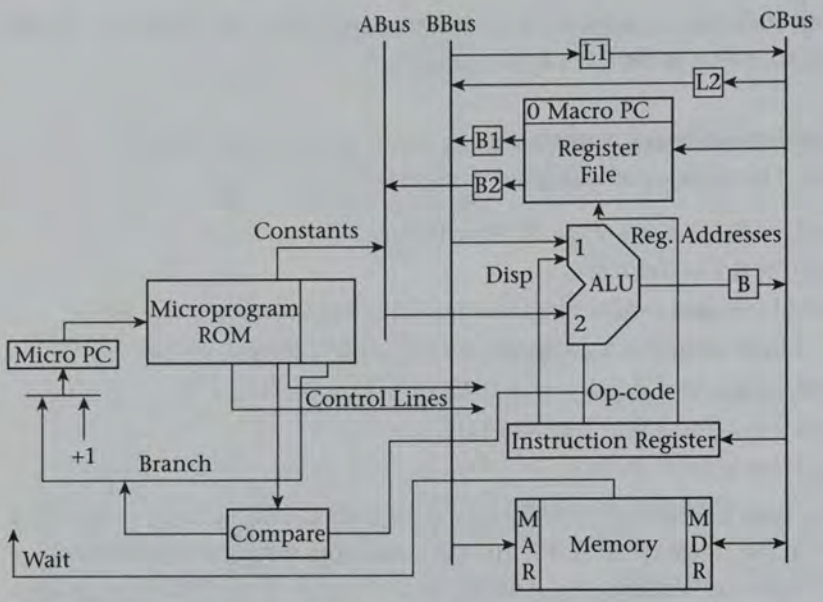


Figure 5.9 Microprogrammed control

Errors in or additions to the macro ISA could be accommodated by simple programming changes, not rewiring with yellow wire as was the current practice with hardwired control. Because of the high cost of ROM in the 1950s, microprogramming was not applied until the introduction of the IBM S360 family in the 1960s (Hellerman 1967). All members of this computer family were microprogrammed except for the highest-performance model.

Figure 5.2, showing the data path with a hardwired control state machine, is redrawn in Figure 5.9 to incorporate microprogrammed control. There are a number of important points to be noted in this figure. One can, in most cases, identify a microprogammed processor because there are two program counters: the micro PC and the macro PC. In some rare designs, a micro PC is not used and the microprogram is in the form of a linked list, a design technique mentioned in Chapter 3.

The micro PC is similar to the macro PC. Its normal sequence is to be incremented for each microinstruction executed. However, a taken microprogram branch will cause the micro PC to be loaded with a branch address from the microprogram ROM. Branches are taken when a field in the current microprogram instruction is compared with an input field with a true result. The fields may be 1 or more bits. The initial bootstrap value in the micro PC points to the instruction fetch microcode segment.

Microinstruction Set Architecture

We now discuss the technique for designing the micro ISA for the interpretation of macroinstructions. The micro-control unit interprets the macroinstructions by means of another program, called a microprogram, which is in the language of the micro ISA. As we will see, the micro ISA is unusual because the requirements for the interpretation

Petitioner Exhibit 1053, Page 206 of 334

consist mainly of transfers of bits or words from one register to another. The micro ISA has two instruction types: move and branch.

Move Microinstruction. The micro ISA has a move instruction that transfers information between the various registers, both architected and implemented, of the data path. The first step in the design of the move microinstruction is to identify the sources and destinations. To accomplish this step, we list the transfer paths from Figures 5.5a to 5.5i by using a register transfer language. The form of the move microinstruction is (source)→destination:

Instruction Fetch
Clock 1: (R0) → B1
Clock 2: (B1) → MAR
Clock 3+Wait: (Mem(MAR)) → MDR

Instruction Decode
(MDR) → IR
Case op-code; Add: go to Add
Load: go to load
Store: go to store
Branch: go to branch

Register Select, for ALU operations[54]
Clock 1: R(IR,RAs1) → B1
Clock 2: R(IR,RAs2) → B2

Effective Address [Base + Displacement]
Clock 1: (R(IR, Rbase)) → B2
Clock 2: (B2) → ALU2
(IR, displacement) → ALU1
Clock 3: (B) → CBus
(CBus) → BBus
(BBus) → MAR

Execute[55]
Clock 1: (B1) → ALU1
(B2) → ALU2
Clock 2: (B) → R(IR, Rdest)

Memory Read
Clock 1+wait: (Mem(MAR)) → MDR
Clock 2: (MDR) → R(IR, Rdest)

[54] The terminology (IR,RAs1) means the RAs1 field of the IR register.
[55] The ALU output buffer is implicitly loaded by each microinstruction that uses the ALU.

Petitioner Exhibit 1053, Page 207 of 334

Memory Write

Clock 1: (R(IR, RAs1)) → B1
Clock 2: (B1) → MDR BBus
(BBus) → CBus
(CBus) → MDR
Clock 3+W: (MDR) → Mem(MAR)

Increment PC

Clock 1: (R0) → B1
Clock 2: (B1) → ALU1
+1 → ALU2
Clock 3: (B) → R0

Branch Target Address (PC relative)

Clock 1: (R0) → B2
Clock 2: (B2) → ABus
(ABus) → ALU2
(IR ,Displacement) → ALU1
Clock 3: (B) → R0

Having identified the transfer paths, we have two options for designing the move microinstruction(s):

1. We can tabulate the unique move microinstructions in the list above, (R0) → B2 for example, and have a unique op-code for each. As there are 19 such unique operations, a 5-bit op-code will suffice.
2. Use a single-bit op-code for the two microinstructions move and branch and have a source field and a destination field. This orthogonal approach will use more microinstruction bits but will provide maximum flexibility for expansion and changes.

For this example design, the second, orthogonal, approach is taken. We now list the sources and destinations required by the move microinstruction in Table 5.3. The data path design assumed that the buses do not have registers for staging; a transfer by means of a bus is accomplished by enabling the bus source and the destination ports.

For example, the contents of B1 are transferred to ALU1 by the BBus, the microinstruction Move, B1, ALU1 is used. Note that the fields of the IR are explicitly defined when a field is used as a source: for example, Move, IR, RAs2, B1 moves the RAs2 field of the macroinstruction register to B1. The macro op-code field is wired to the ALU and invoked along with the ALU when there are valid inputs to the ALU.

Branch Microinstructions. The microprogram must be able to branch. Recall from Figure 5.6 that several states have multiple outputs, an example being the decode state. Thus the micro ISA must have branch instructions, both conditional and unconditional. For this design, there are two conditional branches and one unconditional branch instruction plus the special branch when memory signals a wait:

Petitioner Exhibit 1053, Page 208 of 334

TABLE 5.3 SOURCES AND DESTINATIONS

Sources	Destinations
R0	R0
B1	B1
B2	B2
B	MAR
R (IR, RAs1)	ALU1
R (IR, RAs2)	ALU2
R (IR, base)	R(IR, Rdest.)
Mem(MAR)	Mem(MAR)
MDR	MDR
IR, displacement	IR
Constant field	

1. The conditional branch tests the sign of register R1 and branches if positive.
2. The microinstruction also has a constant field that holds the op-codes of the macro ISA. This field is compared with the macro ISA op-codes to implement the case statement for the decode function.
3. An unconditional branch is needed to return to the beginning of a microinstruction sequence.
4. When the memory generates a wait signal, the microprogram counter is quiescent; it remains in its present state for another clock. In other words, the microprogram branches to its present location. Symbolically this branch is

   (PC) W → PC.

Expanded op-codes are used to uniquely define three of the branch types. The branch target address is a direct address allocated to the microinstruction. Direct addressing is chosen because of the time required for making the addition for PC relative branching and the relatively small number of branch target address bits required. The size of the branch target address depends on the expected size of the microprogram.

Microinstruction Format. What is the format of a microinstruction? Because there are only two instruction types (transfer and branch), the op-code is 1 bit. The source, destination, and constant fields have been discussed. The ALU/Mem field is two bits, which enables the ALU to perform the operation specified by the macro ISA op-code and also enables the memory for a read or write cycle:

Op	Source	Destination	Constant	ALU/Mem	Control	Branch Address

The control field specifies miscellaneous functions such as signifying that the branch microinstruction is unconditional. Codes in the control field define the three types of conditional branches and the unconditional branch instructions.

Petitioner Exhibit 1053, Page 209 of 334

EXAMPLE

How can the three functions of the ALU/Mem field be specified with 2 bits?

Solution

There are a number of solutions to this example; one possible solution is

00 Enable ALU
11 Memory read
10 Memory write

Microprogram: The interpreter of the macro ISA can now be programmed in the language of the micro ISA. Figure 5.10 shows the program for two of the

#	Op	Source	Destination	Constant	ALU/Mem	Control	Branch Address	Comments
0	1	R0	B1					Inst. Fetch
1	1	B1	MAR					"
2	1	Mem(MAR)	MDR		M read			"
3	1	MDR	IR					"
4	0			ALU		Comp.	10	Decode
5	0			Load		Comp.	20	"
6	0			Store		Comp.	30	"
7	0			Branch		Comp.	40	"
8								
9								
10	1	R(IR, RAs1)	B1					ALU
11	1	R(IR, RAs2)	B2					"
12	1	B1	ALU1					"
13	1	B2	ALU2		ALU			"
14	1	B	R(IR, RAdest)					"
15	1	R0	B1					Increment PC
16	1	B1	ALU1					"
17			ALU2	+1				"
18	1	B	R0					"
19	0					Uncon.	0	Return to Inst. Fetch
•								
•								
40	0					test sign	46	Branch
41	1	R0	B1					Increment PC
42	1	B1	ALU1					"
43	1		ALU2	+1				"
44	1	B	R0					"
45	0					Unco.	0	Return to Inst. Fetch
46	1	R0	B1					Branch target
47	1	B1	ALU2					"
48	1	IR, RAdisp	ALU1					"
49	1	B	MAR					"
50	0					Uncon.	0	Return to Inst. Fetch
51								

Figure 5.10 ALU and branch microprograms

Petitioner Exhibit 1053, Page 210 of 334

macroinstruction types, ALU and branch. This microprogram follows the state diagram of Figure 5.6. Recall that the first thing to do is to fetch the macroinstruction. This task is accomplished by the microinstructions in locations 0–3.

Following the macroinstruction fetch, the macro op-code is decoded by a sequence of four microbranch instructions. The branch addresses of these four instructions spread out the microinstructions for the four macroinstruction types. Once the microprogram has been debugged, it can be compacted to reduce the size of the microprogram memory.

The ALU macroinstruction type is implemented in locations 10–14, and the macroprogram counter is incremented by the microinstructions in locations 15–18. An unconditional branch returns the microprogram counter to location 0 to start the fetch of the next macroinstruction.

The macrobranch instruction begins with location 40, where the sign bit of R1 is tested. If not true, the program counter is incremented by the microinstructions in locations 41–44. The microinstruction in location 45 returns unconditionally to location 0 to start the fetch of the next macroinstruction.

From Figure 5.10 we can determine the CPI of the two implemented macroinstruction types. The ALU instruction requires 15 + W clocks whereas the branch instruction requires 14 + W clocks for both the taken and the not-taken paths. This performance is poorer than that of the hardwired implementation, which has a CPI for the ALU instructions of 11 + W clocks and 8 + W clocks for the branch instruction.

Let us examine the sources of the loss of performance. This examination will show where design changes could be made that would improve the performance of microprogrammed control.

1. The decode function is performed in one clock with the hardwired state machine. If the microinstruction ISA is implemented with a four-way branch instruction, decoding can be accomplished in one clock.

Branch	Cond 1	Dest 1	Cond 2	Dest 2	Cond 3	Dest 3	Cond 4	Dest 4

2. Two clocks (two microinstructions) are needed to load the MAR in the microinstruction fetch sequence. A transfer path directly from the register file to the MAR would eliminate this extra clock.
3. Two clocks (two microinstructions) are needed to load the register file buffers B1 and B2 for ALU macroinstructions. This task was performed in one clock with the hardwired state machine control because a dual-ported register file is assumed. To save one clock with a microprogrammed controller, a dual-ported register file is needed as well as additional source and destination fields in the microinstruction format.
4. An explicit unconditional microinstruction branch is required at the end of each sequence to return to the macroinstruction fetch sequence. This clock is not required with the hardwired state machine. A design technique that eliminates these

Petitioner Exhibit 1053, Page 211 of 334

clocks is to give every microinstruction branching capability in addition to its primary function.[56]

The microinstruction format used above has the ability to do a few simple functions in parallel. However, increasing the amount of concurrency in the data path permits more functions to be performed in parallel, thereby increasing the speed of the processor. Additional parallelism is illustrated in the four ideas discussed above for improving performance. With increased data path concurrency, the length of the microinstruction must be increased as well. Thus it is not unusual to see microinstructions of 100 bits or more.

Microprogrammed processors with short microinstructions are called vertical microprogrammed because a sequence of microinstructions are required for performing a function. Microprogrammed processors with long microinstructions are called horizontal microprogrammed because one microinstruction can perform several functions in one clock.

EXAMPLE

The frequency of use of macroinstruction classes are ALU-0.30, branch-0.30, load-0.25, and store-0.15. What is the weighted average number of microinstructions required for performing the decode function?

Solution

From Figure 5.10 we see that the test for an ALU instruction is made followed by a test for load, store, and branch. Thus the weighted average number of microinstructions is

$$\begin{aligned}\text{Weighted average} &= (0.3 \times 1) + (0.25 \times 2) + (0.15 \times 3) + (0.3 \times 4) \\ &= 0.3 + 0.5 + 0.45 + 1.2 \\ &= 2.45 \text{ microinstructions.}\end{aligned}$$

Comment

The use of a four-way branch microinstruction will reduce the weighted mean CPI of the processor by almost one and one half clocks or approximately 10%.

Size of the Microprogram

A few comments on the size of the microprogram are in order. For the implementation of an ISA, such as the IBM S360, a number of steps (microinstructions) are required for each of the macroinstructions. The sum of these steps is the number of microinstructions required in the ROM for a given micro ISA design. There is ample evidence that the number of bits required for implementing an ISA is a constant, not the number of steps. This means that

$$\text{bits} \times \text{words [horizontal]} \approx \text{bits} \times \text{words [vertical]}.$$

Table 5.4 shows the organization of the microprogram memory of four models of the IBM S370 (Case and Padegs 1978). The average size of the memory for these four

[56] A popular microprocessor, ARM, uses this technique at the macro ISA level.

Petitioner Exhibit 1053, Page 212 of 334

TABLE 5.4 IBM S370 MICROPROGRAM MEMORY

Model	Words	Bits/Word	Total Bits
135	24K	16	384K
145	12K	32	384K
155	6K	69	414K
165-II	4K	105	420K
		AVERAGE	400K

models is 400K bits. The implication of this observation on microprogram memory is that a designer can trade off the complexity of the data path (more or less concurrency) for performance without significantly changing the number of bits in the microprogram. The organization of the ROM will change, but not the approximate number of bits.

EXAMPLE

We wish to evaluate the cost of microprogramming compared with that of hardwired control. Microprogramming has a fixed cost of 1000 units of chip area and a variable cost of 1 unit of chip area for the ROM to store each microinstruction. Hardwired control has a cost of 5 units of chip for each instruction. What is the break-even cost for these two design techniques?

Solution

The cost for the microprogrammed control, where n is the number of instructions implemented, is

$$\text{cost(M)} = 1000 + n.$$

The cost of the hardwired control is

$$\text{cost(H)} = 5n.$$

Equate these two costs and solve for n to find the break-even point:

$$\text{cost(M)} = \text{cost(H)},$$

$$1000 + 1n = 5n,$$

$$n = 1000/4 = 250 \text{ instructions.}$$

Comments

This model suggests that for rich CISC ISAs, microprogramming was the design of choice, as in the case of the IBM S360 family. Simple ISAs with only a few instructions are economically implemented with hardwired control.

REFERENCES

Bell, C. G. and Newell, A. (1971). *Computer Structures: Readings and Examples*, McGraw-Hill, New York.

Petitioner Exhibit 1053, Page 213 of 334

Case, R. P. and Padegs, A. (1978). "Architecture of the IBM System/370," *Communications of the ACM*, **21**:1, pp. 73–96.
Hellerman, H. (1967). *Digital Computer System Principles*, McGraw-Hill, New York.
Wilkes, M. V. (1951). "The Best Way to Design an Automatic Calculating Machine," *Manchester University Computer Inaugural Conference*, published by Ferranti Ltd., London.

EXERCISES

5.1 For the example hardwired design, write the input equations using T and JK flip flops.

5.2 For hardwired control, assume that the increment PC states can be subsumed by the instruction fetch states and that the target address state is subsumed by the EA execute state if the clock is lengthened by 10%:
a. Draw the state transition flow-chart.
b. Derive the input equations for JK flip flops.
c. What is the speedup of this implementation over the eight state implementation?

5.3 Explain why a hardwired control unit was used with the fastest model of the IBM S360 whereas microprogrammed control was used with the lower-performance models.

5.4 An alternative way to design a hardwired state machine is to use one flip flop per state rather than encoding the states as shown in the example. Using T flip flops, write the input equations for this design technique.

5.5 Consult the library and find evidence that the number of bits of a microprogram for a given ISA is approximately equal regardless of the speed of the processor. There is available information on the IBM S360 and DEC VAX. You must look.

5.6 Write the microprogram for the macroload instruction.

5.7 Because most macroinstructions require several microinstructions for executing, what happens to the microprocessor if there is an external interrupt in the middle of the instruction execution? Are there design alternatives?

5.8 Some microprogrammed control units have the ability to call microsubroutines. What changes must be made to Figure 5.9 to provide this facility? Recall the discussion on the necessary facilities for subroutines in Chapters 1 and 3.

5.9 From the state diagram of Figure 5.1, determine the internal register organization (accumulator, register file, stack, etc.) of the processor.

5.10 Table 5.3 shows the state transition truth table for the state diagram of Figure 5.6. Instead of encoding the states with five flip flops, use one JK flip flop per state (23 flip flops). Write the J and K input equations for this control state machine.

5.11 Complete the logic equations for the b, c, d, and e flip flops for the State transition truth table of Table 5.2.

Petitioner Exhibit 1053, Page 214 of 334

5.12 For the micro ISA, what are the operations, data types, and addresses?

5.13 Show how the microprogram of Figure 5.10 can be rewritten to use a common body of code for incrementing the program counter.

5.14 Design the state machine for the control of the von Neumann ISA described in Chapter 1.

5.15 Explain how one would design a microprogrammed processor without a macro-program counter or a microprogram counters by using linked lists.

5.16 Is the micro control unit of Figure 5.9 a Harvard or a von Neumann architecture? Explain your answer.

5.17 Can the micro control unit of Figure 5.9 execute subroutines? Explain your answer.

Petitioner Exhibit 1053, Page 215 of 334

SIX

PIPELINED PROCESSORS

6.0 INTRODUCTION

Chapter 5 described the design of serial execution model processors that have either hardwired or microprogrammed control. In this chapter, the techniques used in pipelining are described. Pipelining, sometimes called assembly line processing, has its roots in manufacturing technology. Auto pioneer Henry Ford is generally credited with developing the assembly line for constructing the Model T automobile. There is an unverified story that Ford visited a packing plant in Chicago and saw the disassembly of beef cattle. He is said to have observed that if one can take something apart on a disassembly line then one can do the reverse. Thus the germ of an idea was planted.

A pipelined processor has functional units for each function performed in the interpretation of an instruction. In some cases, the functional units perform more than one operation. Instructions flow into the pipeline and results flow out. The transit time (latency) through the pipeline may be long, but in most cases it is the rate of flow through the pipeline that is important. This is also true of the pipeline that brings water into your home. You don't care how many miles or days it took the water to get to you from the lake; you are only concerned that the flow rate is sufficient.

Base Pipelined Processor

As discussed in Chapter 5 and shown in Figure 5.4, the steps required for interpreting an instruction of the load/store ISA are as follows:

1. Fetch instruction.
2. Decode instruction.
3. Read registers.
4. Execute.
5. Compute effective addresses for data and branch target.
6. Read or write memory.
7. Write to register.
8. Test for a branch.
9. Increment program counter.

Petitioner Exhibit 1053, Page 216 of 334

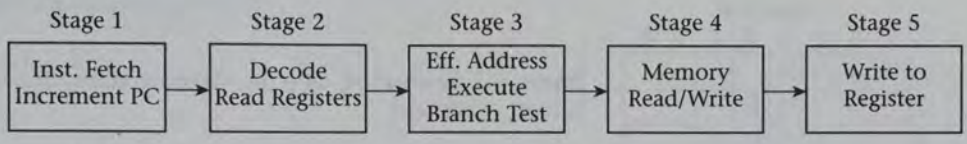


Figure 6.1 Base pipeline

These functions can be grouped into pipeline stages, as illustrated in Figure 6.1, a pipeline similar to the MIPS R2000 pipeline. This pipeline is chosen because of its simplicity; more modern pipelines such as the Intel Pentium Pro and the IBM S390 are quite complicated and do not serve as well for instructional purposes. In the discussions to follow, this pipeline will be referred to as the base pipeline.

Stage 1 performs the instruction fetch and increments the program counter for the next instruction fetch. Stage 2 also performs two functions: it decodes the instruction and fetches the source operands from the register file. Stage 3 performs one of three functions: computes the memory effective address if the instruction is a load, store, or branch; executes the integer operation; or performs the branch test. Stage 4 performs the memory read or write if the instruction is a load or store. Stage 5 writes the results of an integer operation to the register file or writes the output of a load instruction to the register file.

Note that the 9 functions map onto 5 stages. This is possible because some functions are not dependent (they can be done at the same time); for example an instruction can be fetched and the program counter incremented at the same time. In addition, some functions are not used for a particular instruction. For another example, an effective address calculation is not needed with a register add instruction, thus these functions can use the same arithmetic unit.

6.1 PERFORMANCE MODELS

As noted in the introduction, the rate of flow of instructions through the pipeline is the processing rate. The processing rate is usually measured in instructions per unit of time. Because the clock of a processor is time based, processing rate is measured in instructions per clock (IPC) or its reciprocal, clocks per instruction (CPI). IPC and CPI are equal to 1 when the work is flowing smoothly through the pipeline without delays.[57] However, there are events that cause delays or stalls in the pipeline, and CPI is defined as

$$\text{CPI} = 1 + \text{delays}.$$

The primary goal of the designer of a pipelined processor is to have delays that approach zero. Section 6.3 of this chapter describes the source of delays and the techniques for reducing them.

Pipelines present a problem in performance modeling and analysis because actions take place in a space/time relationship. This modeling and analysis problem is best

[57] Superscalar processors, discussed in Section 6.5, have a steady-state CPI less than 1.

Petitioner Exhibit 1053, Page 217 of 334

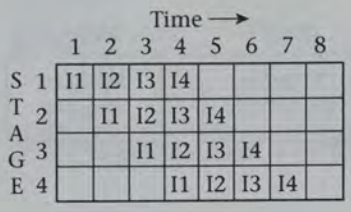


Figure 6.2 Reservation table by stages (courtesy Jones and Bartlett Publishers)

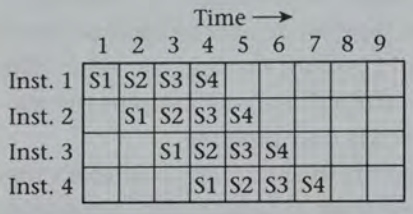


Figure 6.3 Reservation table by instructions (courtesy Jones and Bartlett Publishers)

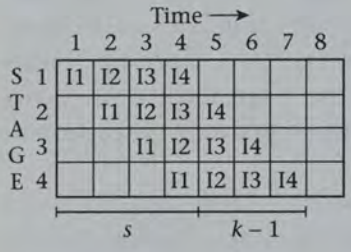


Figure 6.4 Pipeline model reservation table

solved by use of reservation tables (Davidson 1971). Two commonly used forms of reservation tables are shown in Figures 6.2 and 6.3. Figure 6.2 shows time, in clocks, moving from left to right, and the stages of the pipeline are shown in the vertical axis. One can follow an instruction down the pipeline. For example I1 starts in Stage 1 at t1 and is in the last stage, S4 at t4. At t4, four instructions, I1, I2, I3, and I4, are shown in the pipeline. A variation of this type of reservation table is used in manufacturers' literature to show the activity on memory interfaces, bus activity, and the like. Instead of stages, these drawings show various signal lines or signatures.

The other form of reservation table also uses time as the horizontal axis whereas the vertical axis displays the instructions, as shown in Figure 6.3. One follows the progress of an instruction with this reservation table by reading horizontally. Instruction I3 is in Stage 1 at t3, then Stage 2 at t4, and so on. This form of reservation table is used in manufacturers' literature to show the progress of instructions through a pipeline.

The first form of reservation table is prefered for a number of reasons. This form, being similar to what would be seen with a logic probe, seems more natural to electrical engineers. In addition, the second form has a practical problem in drafting: As the number of instructions followed increases, the chart grows in two dimensions whereas the first form grows in only one dimension.

Using reservation tables, we now derive the basic performance model of a pipeline as shown in Figure 6.4. A four-instruction sequence is broken by some event such as a taken branch or another type of delay. To generalize, the length of the sequence is k and the number of stages in the pipeline is s. Thus the number of clocks required for

Petitioner Exhibit 1053, Page 218 of 334

executing this sequence is

$$s + k - 1,$$

and

$$\text{CPI} = \frac{s+k-1}{k} = 1 + \frac{s-1}{k} = 1 + \frac{1}{k}(s-1).$$

Note that this is the same form given early in this section: CPI = 1 + delays. By examining the limits of this model, we gain insight into the limits of pipeline performance:

As $k \to \infty$, CPI $\to 1$,
and as $k \to 1$, CPI $\to s$;
thus $1 \leq \text{CPI} \leq s$.

This means that CPI can be as bad as s (which may suggest the design of short pipelines) or as good as 1 if the sequence of unbroken instructions is long enough.

A serial execution model processor requires a number of steps for completing one instruction before another instruction can begin. A pipeline of s stages is functionally equivalent to a serial processor that requires s steps.[58] Thus the speedup of a pipelined processor compared with that of the serial execution model processor is

$$\text{speedup} = \frac{\text{number of clocks (serial)}}{\text{number of clocks (pipelined)}} = \frac{sk}{s+k-1} = \frac{s}{1+\frac{s-1}{k}}.$$

As $k \to \infty$, speedup $\to s$,
and as $k \to 1$, speedup $\to 1$;
thus $1 \leq \text{speedup} \leq s$.

From this speedup model we can conclude that we may want to design a pipeline with a large number of stages – the opposite conclusion from the models presented previously. What then is the proper length of a pipeline?

6.2 PIPELINE PARTITIONING

The models developed above seem ambivalent regarding the number of stages in a pipeline. To resolve this issue we will subdivide a pipeline into more and more stages, see if there is an optimum number of stages, and find an answer to this design problem.

Start with a one-stage pipeline, as shown in Figure 6.5. If there are enough logic levels between two registers, the complete interpretation of an instruction can be done in one clock. The clock period will be long, but one clock is all that is needed. Assume that the logic path can be divided into two stages. Because the length of the logic path is now $L/2$ gates, the clock period can be reduced by 1/2, and we have a two-stage pipeline with an additional register. The pipeline can then be divided into more and

[58] Note that the average number of states shown in Table 5.1 is 5.2. This is close to the number of stages in the basic pipeline of Figure 6.1.

Petitioner Exhibit 1053, Page 219 of 334

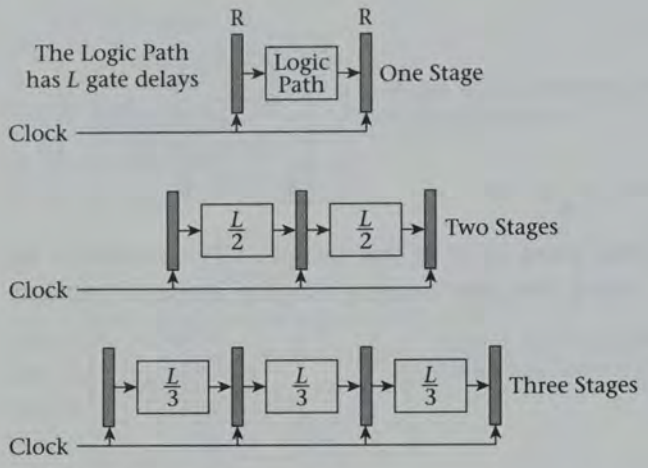


Figure 6.5 Pipeline partitioning

more stages, as shown for $L/3$. The general case is to divide the pipeline into s stages. What then is s for maximum performance of the pipeline?

The execution time of a sequence of instructions is the number of clocks $\times$ clock period $= (s + k - 1)$ clock period. What is the clock period? A first approximation is that the clock period (in gate delays) is equal to L/s because the single stage of length L has been subdivided into s stages.[59] However, there is a practical problem with this approximation.

For performance purposes, latches are used for the registers rather than memory elements such as JK flip flops. The use of latches leads to a number of problems in circuit design. First, all the logic paths in a stage must be of equal length (equal delay) so that the logic inputs arrive at all latches of the register at the same time. Second, the clock has to be distributed to all the register latches. Because of wiring problems on the chip, the clock distribution paths to the register latches will not have equal delay, resulting in clock skew (Kogge 1981).[60] Last, the latches have setup time that must be accommodated in the clock period.

Figure 6.6 shows the logic path between two latches. The clock is distributed from a common source, assumed to be placed very near to latch 1. The logic propagation delay L/s plus any output drive delays is $t(\max)$. The clock for latch 2 must be of sufficient width for the setup time of the latch; this time is w. Because of the delays in clock distribution, there can be skew time α between the clock of latch 1 and latch 2. Thus the minimum clock period is

$$T(\text{clock}, \min) = t(\max) + \alpha + w,$$
$$t(\max) = L/s.$$

[59] Division of L into equal parts may not always be possible; dummy logic stages must be added to the short stages to ensure error-free clocking.

[60] The first recognition known to the author of the clock skew problem was with the MARK I in 1942. The electromechanical switches were driven from a common shaft with independent gears. The subdivision of one rotation of the shaft into cycles had to accommodate the backlash in the gears and the delay in energizing clutches.

Petitioner Exhibit 1053, Page 220 of 334

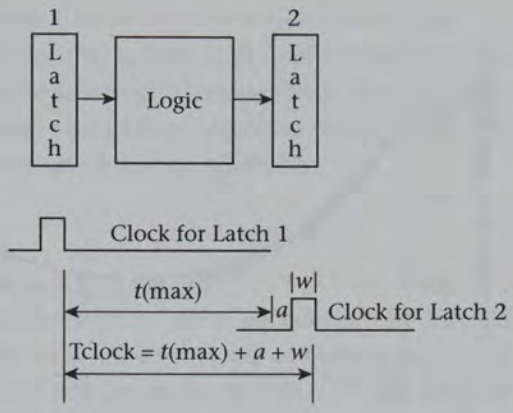


Figure 6.6 Clock period with skew and setup time

The sum of clock skew α and the latch setup time w is τ. Thus the execution time for a multistage pipeline is

$$(s + k - 1)\left(\frac{L}{s} + \tau\right).$$

To find the optimum number of pipeline stages, we take the derivative of the execution time with respect to s, set to zero, and solve for the optimum value of s.

$$s_{opt} = \sqrt{\frac{L(k-1)}{\tau}}.$$

If the value of τ can be reduced, the optimum number of pipeline stages can be increased, thereby increasing the performance by reducing the execution time of a pipelined processor.

EXAMPLE

For a single-stage pipelined processor with $L = 128$ and an instruction sequence $k = 4$, plot the execution time for $s = 1$ to 128 for $\tau = 2$ and 4.

Solution

The solution to this problem is found by applying the parameters to the pipeline execution time equation. The base 2 logarithm of the number of stages is used in Figure 6.7.

Comments

Figure 6.7 shows that the optimum number of pipeline stages is approximately 16 for $\tau = 2$. The minimum is shallow, so finding the exact number of stages is not within the accuracy of this model. Early RISC processors had 4–5 pipeline stages; they are described as underpipelined because the number of stages is smaller than the stages for maximum performance. Most modern pipelined processors have 10–16 stages. Also shown is the reduction in processing time that can be achieved by reducing the value of τ from 4 to 2. Reducing clock skew on VLSI circuits is a major design issue today.

Petitioner Exhibit 1053, Page 221 of 334

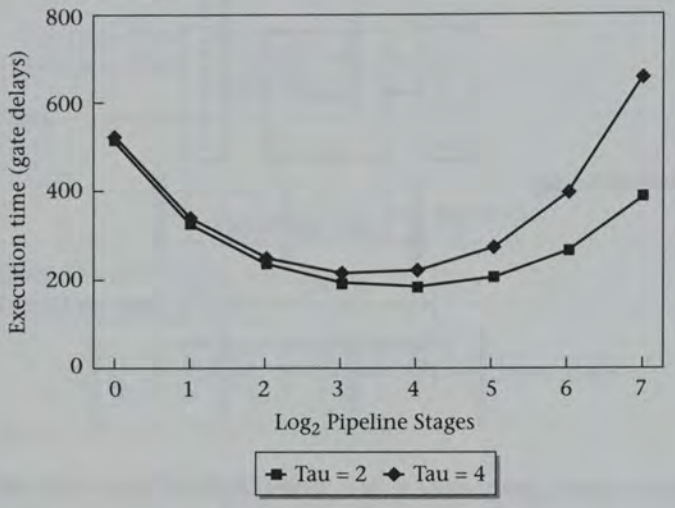


Figure 6.7 Optimum number of pipeline stages

Another design consideration regarding partitioning of a pipeline into stages is balancing the logic level in the stages. Stage 3 of the base pipeline performs two functions: decoding and reading the registers. Other pipelines, such as the Intel Pentium integer pipeline, decode in one stage and select the operand registers in the execution stage. The designer has great latitude in making pipeline partitioning. The design issue is that, to the greatest extent possible, the logic paths of all stages should be equal.

Because all the registers are clocked from a common source, the clock period is the maximum of the sum of individual stage logic delays and τ:

$$\text{clock period} = \text{MAX}[(L1 + \tau), (L2 + \tau), \ldots, (Ls + \tau)].$$

EXAMPLES

1. A processor is partitioned into a pipeline of five stages. The number of logic levels per stage is 8, 13, 10, 11, 12, and τ is 2 gate delays. What is the clock period for this processor in gate delays?

Solution

The clock period is

$$\text{MAX}[(8 + 2), (13 + 2), (10 + 2), (11 + 2), (12 + 2)] = 15 \text{ gate delays.}$$

2. Given the pipeline from the example above, the designer can divide the second-stage logic path into two stages of 6 and 7 gate delays. Using the base pipeline model with pipeline delays of $s-1$ and a sequence of four instructions for $k = 4$, is this a wise design?

Solution

The new clock period is MAX(10, 8, 9, 12, 13, 14) = 14 gate delays. The execution time, in gate delays, for each of the designs is

$$\text{execution time} = (s + k - 1) \text{ (clock period)},$$

$$\text{execution time (first design)} = (5 + 4 - 1)15 = 120 \text{ gate delays},$$

$$\text{execution time (second design)} = (6 + 4 - 1)14 = 126 \text{ gate delays}.$$

Petitioner Exhibit 1053, Page 222 of 334

Comment

This is an example of exploring design alternatives with analytic models. The first design is slightly faster and may be selected. However, recognize the limits of these models, and if there is some compelling implementation reason for using one or the other design, that design should be selected.

6.3 PIPELINE DELAYS

We now return to the problem of delays and how to reduce their effects. The models we have used to this point have had a delay of $s - 1$ at the end of each sequence of k instructions. This delay is the worst case for any processor design and will serve as a starting point for the following discussions. Delays with a pipelined processor come from the following five sources.

1. Memory delays: the latency of a memory reference is greater than one clock. Chapter 3 addressed the control of memory delays.
2. Branch delays: a change of program control to a noncontiguous instruction.
3. Structural hazard delays: the processor must stall because two or more instructions are competing for the same hardware resource.
4. Data hazard delays: an instruction stalls because of data dependencies between instructions.
5. Interrupt delays: the program is interrupted, the interrupt handler is placed in execution; when the handler completes, the program counter control is returned to the interrupted program.

As noted above, in the steady state the pipeline processes one instruction per clock. However, there are various delays that must be added to the steady-state time. There are two aspects of delays: the probability that an instruction will encounter or cause a delay, P_i, and the magnitude of the delay when it is encounted or caused, D_i. The pipeline CPI model was given as

$$\text{CPI} = 1 + \frac{1}{k}(s - 1).$$

In this form, $1/k$ is the probability that an instruction can result in a delay $(s - 1)$. To generalize,

$$\text{CPI} = 1 + \sum_{i=1}^{n} P_i D_i.$$

This simple linear combination of delays overstates the impact that delays have on a pipeline. For example, a cache delay may hide or cover a data hazard delay. Nevertheless, this model does provide insight into the effects, both singularly and in combination, of delays on pipeline performance. In general, this model gives a worst-case estimate of CPI.

Petitioner Exhibit 1053, Page 223 of 334

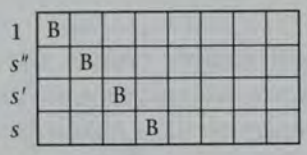


Figure 6.8 Branch strategy reservation table

6.3.1 BRANCH DELAYS

A conditional branch instruction will result in one of two outcomes: (1) the branch is not taken and the in-line instruction sequence continues, or (2) the branch is taken and a change in the value of the program counter starts another sequence. The outcome of a conditional branch will not be known until late in the pipeline. If the instruction sequence is to be changed, a number of instructions will have been partially processed and will have to be abandoned, causing a significant delay.

Two parameters are used in evaluating branching strategies. The first is the probability that an executed instruction is a branch, P_b. The second is the weighted average branch delay (WABD) for taken and not-taken branches. The CPI of the processor with branch delays is

$$\text{CPI} = 1 + P_b \,\text{WABD}.$$

The goal of branch strategy design is the reduction or elimination of WABD, thereby making CPI approach 1. The parameter P_b is determined by the executing program and cannot be influenced by the design of the branch strategy.

Figure 6.8 shows how we use reservation tables to analyze branching strategies. For this analysis, we assign the following symbols to the pipeline stages that play a role in branch strategy design:

Stage 1: the instruction fetch stage
Stage s'': the decode stage
Stage s': the stage at which the effective address of the target is known
Stage s: the stage at which the branch outcome is known.[61]

The relationship for these stages is $s'' \leq s' \leq s$ when numbering the stages from 1, the instruction fetch stage. The symbol B indicates the flow of a branch instruction down the pipeline. For example, the base pipeline of Figure 6.1 has $s'' = 2$, $s' = 3$, and $s = 3$. The branch instruction, denoted B, starts at stage 1 and progresses to stage s.

Pipeline Freeze Strategy

A study of executing branch statistics shows that approximately 60% of all branch instructions are taken and 40% are not taken – continuing in the sequence. We define P_{bt} to be the probability that a branch will be taken. Based on the statistics just cited, $P_{bt} = 0.6$, and $1 - P_{bt} = 0.4$. The parameters P_b and P_{bt} are called the workload parameters in the following paragraphs.

[61] It is not necessary that s'', s', and s be different stages. The Berkeley RISC I had $s'' = s' = s = 2$.

Petitioner Exhibit 1053, Page 224 of 334

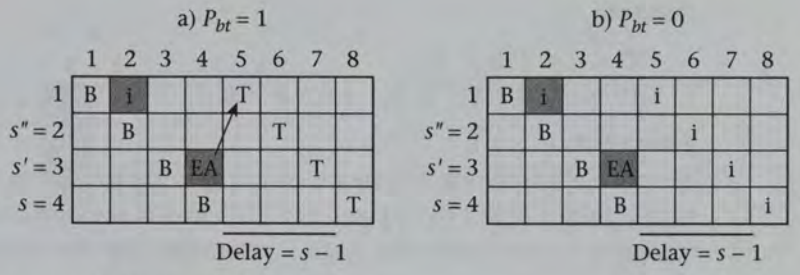


Figure 6.9 Pipeline freeze strategy

The simplest branch strategy that we can think of is to freeze the pipeline when a branch instruction is decoded in s''. Fetching of instructions is halted until the outcome of the branch is known at s. This strategy is shown in Figure 6.9.

We want to determine the WABD of this strategy so that the CPI can be determined. There are two cases to consider: case a), in which the branch is taken ($P_{bt} = 1$); and case b), in which it is not taken ($P_{bt} = 0$).

Case a) $P_{bt} = 1$ shows how the branch strategy works when the branch is taken. The branch is decoded in s'' at t2. In addition at t2, the in-line instruction i following the branch B has been fetched. Because the decoded instruction is a branch, instruction i is abandoned (indicated by shading) and no further instructions are fetched while the branch instruction works its way down the pipeline. At t3, the effective address is computed in s' and saved in a buffer (a buffer is not required if $s' = s$). At t4, the outcome of the branch is known to be taken and the target instruction T is fetched with the effective address in the buffer, which is then cleared. The delay for this case is $s - 1$ clocks.

In like fashion we look at case b), $P_{bt} = 0$, when the branch is not taken. When the not-taken outcome of the branch is know at t4, instruction i is refetched and the in-line stream starts down the pipeline. The delay for this event is $s - 1$.

For the freeze branch strategy the WABD is

$$\text{WABD(freeze)} = P_{bt}(s - 1) + (1 - P_{bt})(s - 1) = s - 1.$$

The probability that an instruction is a branch instruction has been defined as P_b. Thus the CPI of a pipeline by use of the freeze strategy is

$$\text{CPI(freeze)} = 1 + P_b \text{ WABD} = 1 + P_b(s - 1).$$

For this strategy, the taken and the not-taken branches have the same delay, $(s - 1)$. Thus this model is the same as the basic model developed previously from Figure 6.4 in which $1/k = P_b$.

EXAMPLES

1. Consider a pipeline with $s'' = 1$, $s' = 3$, and $s = 4$. The workload characteristics are $P_b = 0.3$ and $P_{bt} = 0.65$. What is the CPI of this pipeline?

Petitioner Exhibit 1053, Page 225 of 334

Solution

$$\text{CPI} = 1 + P_b(s - 1),$$
$$\text{CPI} = 1 + 0.3(4 - 1) = 1 + 0.9 = 1.9.$$

Comment

Because of branching, the steady-state performance of this pipeline has degenerated from CPI = 1 to CPI = 1.9, a 90% loss in performance.

2. If stage s is combined with stage s', requiring that the clock be lengthened by 20%, is performance improved?

Solution

$$\text{CPI} = 1 + 0.3(3 - 1) = 1.6,$$
$$\text{execution time} = \text{CPI} \times \text{clock period} = 1.6 \times 1.2 = 1.92.$$

There is a reduction in performance; $S = 1.9/1.92 = 0.989$.

Comment

The difference in performance is very small and the design could go either way, depending on the actual increase in the clock period.

Predict Take Strategy

The freeze strategy is a very poor strategy from the point of view of performance. Can we do better? Perhaps we can take advantage of the knowledge that approximately 60% of all branches are taken and wire this knowledge into the hardware. The reservation table for such a strategy is shown in Figure 6.10. The case for the branch being taken is shown on the left, case a). When a branch is decoded at stage s'', the fetching of the in-line instructions is terminated and the instructions already fetched are abandoned. When the effective address is known in s' at t3, the target is fetched on the next clock, t4, before the outcome of the branch is known.[62] We do this because we are predicting that the branch will be taken. When this prediction is correct, the delay is $s' - 1$.

When the branch is not taken, as shown by case b), not only are the in-line instructions abandoned but the prefetched target instruction is as well, because the prediction

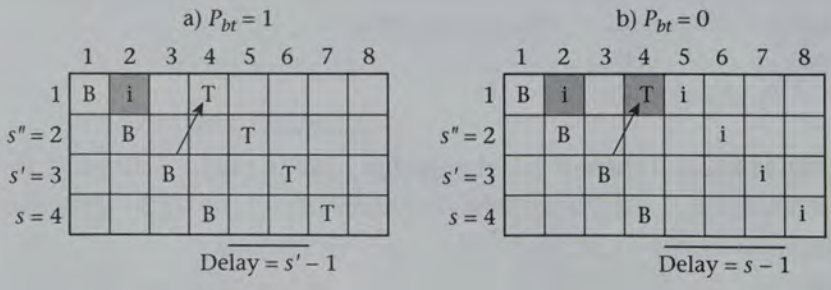


Figure 6.10 Predict take strategy

[62] Called speculative execution.

Petitioner Exhibit 1053, Page 226 of 334

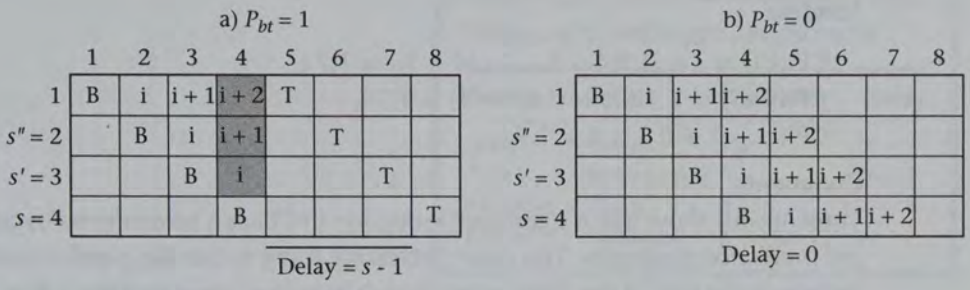


Figure 6.11 Predict not-take strategy

was wrong. The first in-line instruction must be refetched. The delay for this case is $s - 1$.

We can now derive the WABD and CPI for the predict take strategy:

$$\text{WABD(PT)} = P_{bt}(s' - 1) + (1 - P_{bt})(s - 1) = s - 1 - P_{bt}(s - s'),$$

and

$$\text{CPI(PT)} = 1 + P_b \text{ WABD} = 1 + P_b[s - 1 - P_{bt}(s - s')].$$

Predict Not-Take Strategy

Why not investigate a branch strategy that predicts that the branch will not be taken, the opposite of the predict take strategy? The reservation table for this strategy is shown in Figure 6.11.

As the hardware is predicting that the branch will not be taken, the strategy speculatively fetches as many in-line instructions as possible; $(s - 1)$ or three instructions in this example. The hardware waits until the outcome of the branch is known at stage s before fetching the target instruction. The branch target address must be saved in a buffer with this example. If the prediction is wrong and the branch is taken, as shown for case a) of Figure 6.11, the delay is $s - 1$. However, if the branch is not taken as predicted, the delay is zero, as shown with case b).

The WABD and CPI are derived as

$$\text{WABD(PNT)} = P_{bt}(s - 1) + (1 - P_{bt})0 = P_{bt}(s - 1),$$

and

$$\text{CPI(PNT)} = 1 + P_b P_{bt}(s - 1).$$

EXAMPLE

Use the pipeline and workload data from the prior example, compute the CPI for the predict take and predict not-take strategies, and compare them with the CPI for the freeze strategy.

Petitioner Exhibit 1053, Page 227 of 334

Solution

$$\text{CPI(PT)} = 1 + 0.3[4 - 1 - 0.6(4 - 3)] = 1.72,$$
$$\text{CPI(PNT)} = 1 + 0.3 \times 0.6(4 - 1) = 1.54,$$
$$\text{CPI(F)} = 1 + 0.3 \times 3 = 1.9.$$

Comments

These results show that a significant reduction in CPI can be achieved by applying prediction strategies. The most interesting thing is that the predict not-take strategy is the best of the three, even though branches are most often taken. The reason for this is that there is a big win, zero delay, for the not-taken case that constitutes ~40% of all branches. This result seems to be counter to Amdahl's law that says to speed up the most frequent case. However, if the event probabilities are approximately equal and a major reduction in the time of the smaller event probability can be achieved, then that design option should be explored; it may be the best design option.

General Prediction for Branch Target Buffers and Caches

The models developed above are based on the idea that the prediction, either taken or not taken, is a designer's choice and is wired into the processor. Is it possible to be smarter than this and have the prediction adjusted dynamically as the program executes? The answer is yes: dynamic prediction is a feasible design option and can further reduce the CPI of a processor.

When looking at the two strategies above, we see that with a dynamic prediction system there are four possible events. These events and the delays from Figures 6.10 and 6.11 are tabulated in Table 6.1.

A perfect predictor (a predictor that correctly predicts both the taken and the not-taken branches) will ensure that only the first and the fourth events will ever occur; the second and the third events will never occur. The accuracy of prediction is called A_p, and for a perfect predictor, $A_p = 1$.

The event probabilities with perfect prediction are shown in Table 6.2. Because the predictor is perfect, the probability of the first event is P_{bt} and that of the fourth event is $1 - P_{bt}$. The other two events have zero probability. This leads to the WABD model

$$\text{WABD}(A_p = 1) = P_{bt}(s' - 1) + (1 - P_{bt})0 = P_{bt}(s' - 1),$$

TABLE 6.1 GENERAL PREDICTION EVENTS AND DELAYS

Predict	Actual	Delay
Take	Taken	$s' - 1$
Take	Not taken	$s - 1$
Not take	Taken	$s - 1$
Not take	Not taken	0

Petitioner Exhibit 1053, Page 228 of 334

TABLE 6.2 PERFECT PREDICTION

Predict	Actual	Probability	Delay
Take	Taken	P_{bt}	$s' - 1$
Take	Not taken	0	$s - 1$
Not take	Taken	0	$s - 1$
Not take	Not taken	$1 - P_{bt}$	0

and

$$\text{CPI}(A_p = 1) = 1 + P_b P_{bt}(s' - 1).$$

This result is remarkable in that, with a perfect predictor, the location of stage s' is paramount whereas the location of s, is not important. Thus the designer who wants to design a good branch strategy to reduce CPI has two tasks:

1. To devise a good predictor.
2. To place s' early in the pipeline (if $s' = 1$ and $A_p = 1$, then WABD = 0).

Remember that there can be a trade-off here. Stages can be combined to move s' forward, reducing CPI at the expense of a longer clock period. This may or may not improve performance, but the effect can be modeled. We will consider other techniques for placing s' early in the pipeline in later paragraphs.

Dynamic Branch Prediction

The first requirement for a high-performance branch strategy is a good branch predictor. In this section we discuss branch predictors that dynamically modify the prediction as the program executes. These dynamic techniques adjust the prediction on each unique branch instruction as it is executed. There are three general forms of dynamic branch prediction: (1) predict based on branch instruction characteristics, (2) predict based on target address characteristics, and (3) predict based on history. See Cragon (1991) for a more complete discussion on branch prediction.

Branch Instruction Characteristics

A straightforward branch instruction characteristic is to provide a bit in the branch instruction that can be set by the programmer/compiler. This bit will instruct the processor as to the most likely direction of the branch; the bit is not modified during execution. For example, if a loop is being programmed, the branch direction bit will be set to take the branch. If the loop is executed 100 times, the prediction will be accurate 99% of the time. A branch prediction bit is used in the PowerPC branch instruction.

The branch instruction op-codes provide insight for making branch predictions at stage s''. For example, an unconditional branch is always taken and should therefore always be predicted as taken. Statistical analysis of branch instruction op-codes shows

Petitioner Exhibit 1053, Page 229 of 334

TABLE 6.3 BRANCH EVENTS AND DELAYS

Branch Type	Strategy	Actual	Probability	Delay
Conditional	PNT	Taken	$(1 - P_{uncb})P_{cbt}$	$s - 1$
Conditional	PNT	Not taken	$(1 - P_{uncb})(1 - P_{cbt})$	0
Unconditional	PT	Taken	P_{uncb}	$s' - 1$
Unconditional	PT	Not taken	0	0
			SUM = 1	

that the conditional branch instructions also have patterns of taken and not taken that can be used for prediction (Lee 1984).

EXAMPLE

Consider a branch strategy based on the branch instruction op-code. All unconditional branches are predicted taken and all conditional branches are predicted not taken. The fraction of all branches that are unconditional is P_{uncb}.

a. What is the WABD for this strategy?

b. Using the statistics of the previous examples and $P_{uncb} = 0.3$, find the CPI of the processor.

Solution

We can approach this problem by constructing a table of event probabilities and their delays. To solve this problem, we define P_{cbt} as the probability that a conditional branch is taken. Recall that P_{bt} is the probability that all branches are taken. In Table 6.3, the four possible states of the branch strategy are listed along with their event probabilities and delays.

From Table 6.3,

$$\text{WABD} = (1 - P_{uncb})P_{cbt}(s - 1) + P_{uncb}(s' - 1).$$

What is the value of P_{cbt}? We can find P_{cbt} by first noting that P_{bt} is the probability that all branches are taken. We then determine the weighted mean of the conditional and the unconditional branch probabilities:

$$P_{bt} = (1 - P_{uncb})P_{cbt} + P_{uncb} \times 1,$$

$$P_{cbt} = (P_{bt} - P_{uncb})/(1 - P_{uncb}).$$

From the data, $P_{cbt} = (0.6 - 0.3)/(1 - 0.3) = 0.428$. We can now compute the WABD and CPI of this strategy:

$$\text{WABD} = (1 - 0.3)0.428(4 - 1) + 0.3(3 - 1) = 1.49,$$

$$\text{CPI} = 1 + P_b\,\text{WABD} = 1 + (0.3 \times 1.49) = 1.44$$

Comments

By predicting on the op-code, the CPI has been reduced to 1.44 from 1.54 for the PNT strategy. The model derivation can be checked by setting P_{uncb} to 0 and 1 to see if the two models for predict taken and predict not taken are obtained.

Petitioner Exhibit 1053, Page 230 of 334

Also by setting $P_{uncb} = 0$, meaning that all branches are conditional, $P_{cbt} = P_{bt}$, as would be expected.

Target Address Characteristic

The branch target address has two characteristics that can be used for branch prediction. The characteristics are the direction of the branch and the magnitude of the branch distance. Short branches and backward branches are characteristic of loops in a program and are usually taken. Long branches and forward branches in a program are used for major changes in program control and are usually not taken.

How are these predictions made? Consider a PC relative branch target address. Determining the direction of a branch is relatively easy as the displacement in the branch instruction is a signed integer. Thus the sign bit of the displacement can be used for prediction.

Determining the branch distance is more difficult. First a threshold value has to be determined, say 108 instructions. Then how is the comparison made between the displacement and the threshold? A subtraction would add significant time to the decode stage. A reasonable design is to set the threshold at an even power of 2 (128, not 108 as suggested above) and examine the bits in the displacement to see if the threshold is exceeded.

Branch History

Statistical studies show that if the history of the branch direction of a given branch instruction is collected in hardware, a prediction can be made the next time that branch instruction is executed. For example, if there is a bit in the branch instruction that is set or reset depending on the outcome of the current execution, that bit is a good predictor for the next execution. A branch that is taken is more likely to be followed by a taken branch than a not-taken branch the next time it is executed. A problem with this scheme is that since the bit is in the instruction, we have to deal with self-modifying code.

A more practical solution is to have a field of history bits in the instruction cache associated with each instruction. These bits are not part of the instruction but are appended in the cache. Research has shown that the more bits in the history field, the more accurate the prediction. To illustrate this, a 1-bit and a 2-bit scheme are shown in Tables 6.4 and 6.5, respectively. The data are from Lee (1984) for a test benchmark program. The first column shows the history and the prediction. The second column shows the outcome in italics. N and T are used rather than not taken and taken. For the 1-bit history, Table 6.4 shows that the pattern NN occurs with the probability 0.309 whereas the pattern NT occurs with the probability 0.071.

The hardware, looking at the history bit, will predict Not if the last occurrence was not taken and Take if the last occurrence was taken. With 1 bit, the accuracy of prediction is $A_p = 0.309 + 0.565 = 0.874$.

A history that uses 2 bits, illustrated in Table 6.5, also uses the Lee data. Read from right to left; for example NT means that the last branch was taken and the one before it was not taken. The probabilities of the correct predictions are shown in boldfaced type.

Petitioner Exhibit 1053, Page 231 of 334

TABLE 6.4 1-BIT BRANCH HISTORY PROBABILITIES

History (Predict)	Next	Probability
N(N)	*N*	0.309
N(N)	*T*	0.071
T(T)	*N*	0.046
T(T)	*T*	0.565

TABLE 6.5 2-BIT HISTORY PROBABILITIES

History (Predict)	Next	Probability
NN(N)	*N*	**0.294**
NN(N)	*T*	0.015
NT(N)	*N*	**0.051**
NT(N)	*T*	0.019
TN(T)	*N*	0.017
TN(T)	*T*	**0.039**
TT(T)	*N*	0.013
TT(T)	*T*	**0.552**

Two takens in a row will predict taken with an accuracy of $0.552/(0.013+0.552) = 0.97$. Two not taken in a row will predict not taken with a prediction accuracy of $0.294/(0.294+0.015) = 0.95$. When the two history bits differ, the prediction is based on the historical statistics of the next execution. That is, NT will predict *N* because $0.051 > 0.019$. The overall accuracy of prediction when a strategy based on this benchmark data is used is $0.294 + 0.051 + 0.039 + 0.552 = 0.936$; a 10% improvement over the 1-bit history scheme.

EXAMPLE

Determine the accuracy of prediction using a 2-bit history scheme that uses the following strategy. If the two history bits are the same (NN or TT), predict *N* or *T*, respectively. If the bits differ (NT or TN), predict based on the last execution: N predict *N*, T predict *T*.

Solution

Table 6.5 is modified in Table 6.6 with the predictions changed for the cases for which the two history bits differ. The probabilities of the correct predictions are shown in boldfaced type.

The accuracy of prediction is the sum of the probabilities for which the prediction is correct; $(0.294 + 0.019 + 0.017 + 0.552) = 0.882$.

Comment

The accuracy is only slightly better than the 1-bit (0.874) scheme. The hardware

Petitioner Exhibit 1053, Page 232 of 334

TABLE 6.6 MODIFIED 2-BIT HISTORY

History (Predict)	Next	Probability
NN(N)	*N*	**0.294**
NN(N)	*T*	0.015
NT(T)	*N*	0.051
NT(T)	*T*	**0.019**
TN(N)	*N*	**0.017**
TN(N)	*T*	0.039
TT(T)	*N*	0.013
TT(T)	*T*	**0.552**

required for implementing this scheme is almost as complex as the 2-bit scheme and the prediction accuracy is considerably worse.

Problems exist with all predictors used with a pipelined processor. One significant problem is that the history state of a conditional branch instruction is updated after the outcome of the branch is known, at stage s. If a particular branch instruction is executed with an interval less than s, then the prediction is made with "stale" information. In other words, the last execution has not been posted. A 3-bit history may be only as good as a 2-bit history because of this delay.

Other problems with some prediction schemes, such as the 1-bit scheme of Figure 6.4, is that for pathological branching patterns, such as T, N, T, N, T, . . . , the branch prediction will always be wrong. Also, when there are context switches, it takes some time to reestablish a history and the prediction accuracy may be compromised for a period of time. It is possible to save the history information as part of the processor state when making a context switch.

Reducing s'

Now that we see how to have prediction accuracy that approaches 1, how do we reduce s'? Recall that s' is the stage of the pipeline at which the effective address is known. The solution to this design problem is found in the observation that the target effective address of a branch instruction usually does not change from one execution to the next. In other words, target effective addresses have a form of temporal locality, and caches can be used to save previously computed and used effective addresses. After another execution of the branch instruction, the target effective address can be immediately available and $s' = 1$ and the WABD = 0. There are two forms of these effective address caches: branch target buffers (BTBs) and branch target caches (BTCs).

Branch Target Buffer

A block diagram of a BTB is shown in Figure 6.12. An entry in the BTB consists of three fields: the effective address of the branch instruction, the effective address of the branch target, and history information for prediction. The instruction addresses from the I fetch stage are routed to the instruction cache and to the BTB. The BTB is

Petitioner Exhibit 1053, Page 233 of 334

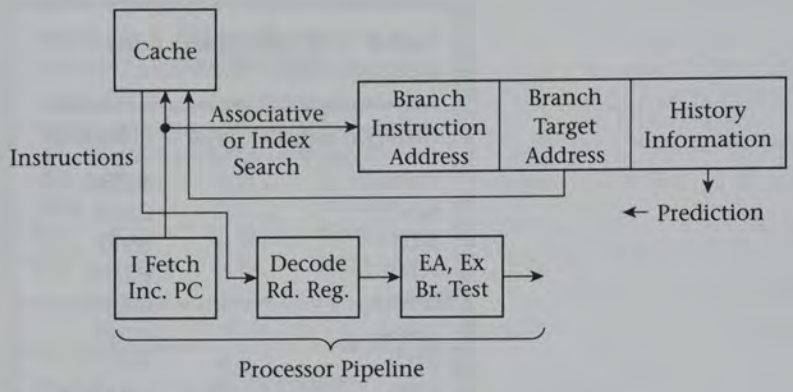


Figure 6.12 BTB

organized and accessed in any of the usual cache access schemes: direct, set associative, or full associative. If there is a hit on the BTB, meaning the branch instruction has been used previously and is still in the cache, and if the history information predicts a taken branch, then the branch target address is routed to the instruction cache and the target instruction is fetched into the decode stage.

Because the BTB access is in serial with the access of the instruction cache, the clock period of the processor may need to be increased or a second clock used, making $s' = 2$. There are a number of design questions with a BTB. What happens if there is a miss on the BTB? What happens if there is a hit on the BTB but the prediction is a not-taken branch? These questions are answered by tabulating the events and deciding on a policy with its delay.

We have to define some parameters. P_{btbh} is the BTB hit probability, P_{pbt} is the probability that the prediction is to take the branch, and A_p is the accuracy of prediction. Table 6.7 shows the events, delays, and default strategies for the behavior of the BTB. This table draws from Table 6.1 for the default delay when the prediction is wrong. When there is a BTB miss, a pipeline freeze strategy is assumed.

From Table 6.7, we can write the equation for the WABD. Note that for two events the delay is zero, and for all the other events the delay is $(s - 1)$:

$$\text{WABD} = [P_{btbh}P_{pbt}(1 - A_p) + P_{btbh}(1 - P_{pbt})(1 - A_p) + (1 - P_{btbh})](s - 1),$$
$$\text{WABD} = [1 - (P_{btbh}A_p)](s - 1).$$

TABLE 6.7 BTB EVENTS AND DELAYS

BTB	Prediction	Actual	Probability	Delay
Hit	Take	Take	$P_{btbh}P_{pbt}A_p$	0
Hit	Take	Not take	$P_{btbh}P_{pbt}(1 - A_p)$	$(s - 1)$
Hit	Not take	Not take	$P_{btbh}(1 - P_{pbt})A_p$	0
Hit	Not take	Take	$P_{btbh}(1 - P_{pbt})(1 - A_p)$	$(s - 1)$
Miss	Freeze	Don't care	$(1 - P_{btbh})$	$(s - 1)$
			SUM = 1	

Petitioner Exhibit 1053, Page 234 of 334

Note that the location of stage s is now in the WABD model. However, s is important only when A_p is much less than 1. In addition, the size of the BTB can be increased, increasing its hit ratio and thereby mitigating the effects of poor branch prediction or a large s.

The BTB is accessed and the taken or not-taken path is followed based on the prediction. What happens when the prediction is wrong or there is a miss on the BTB? Remember that the outcome of the branch is not known until stage s. Thus there must be a parallel pipeline that executes a default strategy to cover these cases. In other words, this parallel pipeline operates as if the BTB did not exist. For Table 6.7, the default strategy is predict not taken with a delay of $s - 1$.

EXAMPLE

Consider the pipeline example used above with $s = 4$, $P_b = 0.3$, and $P_{bt} = 0.6$. Also assume that $P_{btbh} = 0.9$ and $A_p = 0.9$. What is the WABD and the CPI of this strategy?

Solution

$$\text{WABD} = [1 - (0.9 \times 0.9)](4 - 1) = 0.57,$$
$$\text{CPI} = 1 + (0.3 \times 0.57) = 1.17.$$

Comments

Note that the term P_{pbt} drops out of the model. With a BTB having the estimated parameters above, the CPI of the processor is now approximately 10% less than the theoretical steady state CPI of 1. This example shows the great benefit of a BTB for reducing CPI.

Branch Target Cache

Another solution to the problem of reducing s' is to use a BTC. Why store the target address that accesses the instruction cache as done with a BTC? Why not store the target instructions themselves? This is the approach taken with a BTC. A block diagram of a BTC is shown in Figure 6.13. The default pipeline is shown in this figure. Note that the BTC is similar to the BTB except that the target instruction stream

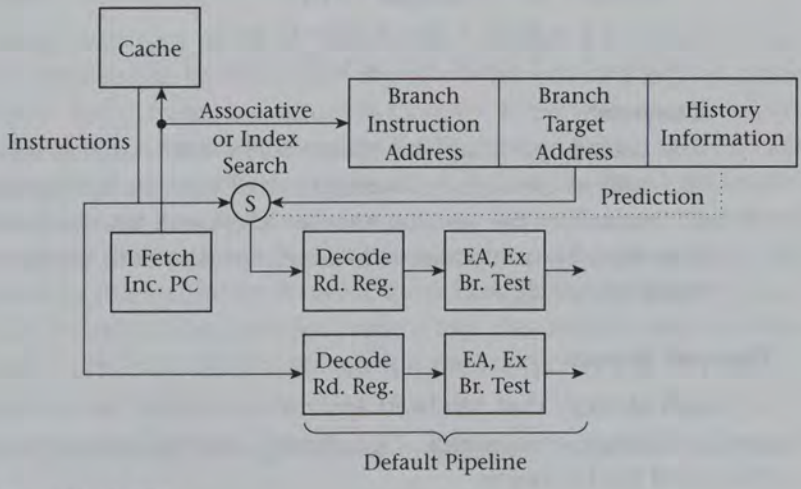


Figure 6.13 BTC

Petitioner Exhibit 1053, Page 235 of 334

is routed into the decode stage when there is a hit and the prediction is to take the branch. This design eliminates the need to lengthen the clock or to provide a cycle delay because of the added delay in accessing the instruction cache. For this strategy $s' = 1$.

A design problem with the BTC is providing a source for the instructions following the target instruction. The solution to this problem is to have the BTC hold the target instruction and the next i target instructions; i is usually 4 or 8. For example, the AM29000 BTC holds a sequence of four target instructions.

A design problem with a BTB or BTC is finding time to update them. There are two types of updates. The first type is needed when there has been a miss (the branch has not been used before) and a complete entry must be formed. The second type of update is needed when the history field must be updated. In both cases, time must be found to update the BTB or BTC that may reduce the performance of the system. Note that not all uses of the BTB or BTC require updating. If 1 bit is used for history, when the prediction is correct, the history does not need to be updated. Saving in update time may compensate for the lower prediction accuracy.

As the BTB and BTC are caches, provisions must be made to evict an entry when the cache is full and a new entry must be created. The same strategies are applied as are applied to instruction and data caches. Also, if the BTB or BTC are set associative, there can be set conflict misses.

EXAMPLE

Assume that an update of a BTB requires one cycle and an update is required for each execution of a branch instruction; the frequency of update is P_b. What is the CPI of the processor if the update cycles are accounted for? Find the CPI by using the parameters of the preceding example.

Solution

The WABD for the BTB, derived in Table 6.7, is modified to account for the update cycle. One cycle is required for the update that is needed for each branch instruction executed:

$$\text{WABD} = (1 - P_{btbh} A_p)(s - 1) + 1,$$
$$\text{CPI} = 1 + P_b[(1 - P_{btbh} A_p)(s - 1) + 1],$$
$$\text{CPI} = 1 + 0.3(0.57 + 1) = 1.47.$$

Comment

The update cycles add 0.3 cycles ($1.47 - 1.17 = 0.3$) to the CPI and are a significant performance hit. A designer may consider having a dual-ported BTB or BTC to reduce the penalty. Another solution to this design problem may be to save the update information in a buffer and perform the update on a BTB cycle that is not a hit.

Delayed Branch

A branch strategy that has been applied successfully to short pipeline RISC processors is called delayed branch. This strategy depends on cooperation between the software and the hardware.

Petitioner Exhibit 1053, Page 236 of 334

The compiler introduces $s - 1$ no-op instructions following each branch instruction.[63] Thus the pipeline is performing no useful work while waiting for the branch to be resolved at stage s. The compiler then attempts to replace the no-ops with useful instruction that can be executed in the no-op slots without destroying the logic of the program. Several studies have shown that, on average, one no-op can be replaced with a useful instruction.

This strategy is useful for pipelined processors with $s \leq 5$. For longer pipelines, too many no-ops are executed, thereby increasing the number of instructions executed for many tasks.

6.3.2 STRUCTURAL HAZARDS

Structural hazards occur when there are two or more demands on a single resource in the same clock period. The very nature of a pipelined processor is concurrency: designing it so that as many activities as possible are being performed at the same time. This concurrency-directed design can lead to the simultaneous access of the same hardware resource.

Consider the function of an ALU in a processor. With a serial execution model processor (discussed in Chapter 5) one ALU can perform the three functions of incrementing the program counter, computing an effective address, and performing the operation needed by the instruction. With a pipelined processor having a number of instructions in simultaneous execution, one ALU will not support the three operations at one time. There are three solutions to this problem that guarantee correct logical execution of the program:

1. Use priority logic to delay all but one of the requests, resulting in delays for the other requesters.
2. Replicate the resource, such as three ALUs, for the example cited above.
3. Incorporate one resource that is fast enough to support all requests in one clock period.

Priority Selection

A number of design situations in which the priority solution is the preferred approach to structural hazards can be cited. One of these is the use of a unified cache as a second-level cache. Recall from an example in Chapter 4 that with split first-level caches, only a small number of memory references (instruction and data) miss and are handled in the second-level cache. The references that miss on the first-level caches are not likely to be correlated; that is, they do not occur at the same time. Thus there is only a small probability that both will miss at the same time and cause a delay that is due to two accesses to the second-level cache. Even though the probability of simultaneous access of the second-level unified cache is low, the designer must provide priority logic to resolve this structural hazard when it does occur.

[63] A no-op instruction does absolutely nothing. There are no side effects except incrementing the program counter.

Petitioner Exhibit 1053, Page 237 of 334

A similar situation exists on the bus that connects a unified cache to real memory that is time shared with I/O. This case is influenced by the write policy of the cache. With a write through data cache, the bus is utilized approximately 50% of the time because all the read and write misses require bus transfers. This case is usually handled by giving priority to the I/O operation because these operations are at a lower frequency than cache misses. And, as discussed in Chapter 7, if some I/O devices are delayed access to the bus, a significant performance loss can occur.

Buses internal to the processor may also use priority selection. The outputs of the floating-point multiplier/divider and adder execution units of the IBM S360-91 share a single bus called the common data bus. This bus routs results to the register file and the execution unit inputs. If both execution units produce a result in the same clock, priority logic delays one result for one clock. The designers of this processor believed that the frequency of occurrence of this event would be small and the resulting delay would thus be small.

EXAMPLE

Figure 6.1 shows that two pipeline stages (stage 2 and stage 5) of the base pipeline can access the register file. Depending on the particular instructions that are flowing through the pipeline at any time, the register file may be accessed with two reads and one write in the same clock. Assume that the register file is dual ported; that is, it can perform two reads in one clock. However, if a write occurs on the same clock, it must be delayed by priority logic. Assume further that on 1/4 of the clocks there are no register file accesses, on 1/4 of the clocks there are two reads, on 1/4 of the clocks there are two reads and one write, and on 1/4 of the clocks there is a write to register file. What is the CPI of the processor, ignoring all other delays?

Solution

Only on 1/4 of the clocks when there are two reads and one write is there a stall on the register file. The structural hazard is resolved by priority logic. For this clock, the pipeline is stalled or delayed one clock:

$$\text{CPI} = 1 + (\text{probability of a delay} \times \text{delay}),$$
$$\text{CPI} = 1 + (0.25 \times 1) = 1.25.$$

Replicated Resources

When the probability of a conflict is such that a significant performance loss will occur by priority selection of a single resource, the resource can be replicated. There are a number of examples of this approach to solving the structural hazard problem. For example, three ALUs are used in some pipelined processors: one for incrementing the PC, another for effective address calculations, and another for the operation execution unit. Note that in these cases the ALUs can be highly specialized to the operation at hand.

EXAMPLE

The base pipeline fetches an instruction every clock and has a read or write operation on load and store instructions. Assume that 0.3 of the instructions are

Petitioner Exhibit 1053, Page 238 of 334

loads or stores. With a unified cache, the load or store instructions must be delayed one clock by priority logic. With split caches, there is no delay. Refer to Figure 4.10 for the two cache topologies. What is the speedup in CPI that results from using a split cache and ignoring cache misses? See Figure 4.10.

Solution

CPI = 1 + (probability of a delay × delay),

CPI (unified cache) = 1 + (0.3 × 1) = 1.3,

CPI (split cache) = 1 + (0.3 × 0) = 1.0,

speedup = 1.3/1.0 = 1.3.

Comment

Adding the second cache gives a significant improvement to the performance of the processor. Because of the relatively low cost of hardware, replication of resources is today a preferred solution to many structural hazard problems.

In some processors, the TLB is replicated so that name translations for instructions and data will not face structural hazards. When storage locations are replicated, care must be taken that the contents of these locations are coherent, as discussed in Chapter 4. Otherwise, the execution of the program may not be logically correct. The designer must consider whether the time lost to maintain coherency may not be greater than the delays that result from a shared resource by using priority logic.

6.3.3 DATA DEPENDENCIES

Data dependencies occur in an executing program when, because of the latency of a pipeline, the program may not execute correctly. These dependencies do not occur in a serial execution model processor. There are three classes of data dependencies:

1. Read after write (RAW), also known as a true dependency
2. Write after read (WAR), also known as an antidependency
3. Write after write (WAW), also known as an output dependency

As will be described in the following paragraphs, these three classes of dependencies are detected by the hardware, and the hardware ensures correct execution by both ad hoc and general techniques.

Read after Write Dependencies

This dependency occurs when an instruction needs an operand that will be produced by a previous instruction, but the result has not yet been produced because of the latency of the pipeline. Consider the following program fragment consisting of two instructions; instruction i is followed by instruction j. There may be zero, one, or more instructions between i and j:

i R1 = R2 + R3,

j R4 = R1 + R6.

Petitioner Exhibit 1053, Page 239 of 334

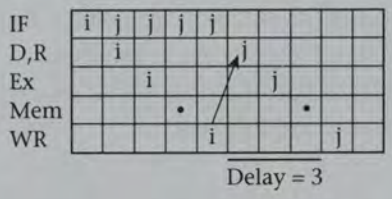


Figure 6.14 RAW dependency

There is a RAW dependency on R1 as instruction j cannot be executed correctly until instruction i has posted its result to the destination register R1. Keep in mind that RAW dependencies are artifacts of the program. A smart compiler can do some rearranging of the program to reduce the frequency of occurrence of RAW dependencies, but all RAW dependencies cannot be eliminated.

The processor designer has one objective: reduce the delays when RAW dependencies do occur. These objectives can be stated in terms of CPI:

$$\begin{aligned}\text{CPI} &= 1 + (\text{probability of a RAW dependency} \times \text{ RAW delay}) \\ &= 1 + P_{raw} D_{raw}.\end{aligned}$$

We now determine the value of the RAW delay. A reservation table is shown in Figure 6.14 for the two-instruction fragment executing on the five-stage pipeline of Figure 6.1.[64] Instruction i is processed down the pipeline with the result written into the register file. Because instruction j needs this result, it is delayed for three clocks until the result can be read from the register file. For many programs, a RAW dependency between adjacent instructions, defined as P_{raw}, has a probability of approximately 0.2. Thus, the CPI of this processor is

$$\begin{aligned}\text{CPI} &= 1 + P_{raw}\ D_{raw}, \\ \text{CPI} &= 1 + (0.2 \times 3) = 1.6.\end{aligned}$$

Because this is a significant reduction in performance, something must be done to overcome the loss. The technique used for reducing or eliminating this performance loss is called forwarding. The last three stages of the base pipeline and a reservation table are shown in Figure 6.15.

A logic path takes the output of the execution unit and introduces it as an input to the execution unit, bypassing the register file. Thus the execution of instruction j can start on the next clock. The result of instruction i is written into the register file normally because that result may be needed by a subsequent instruction. With forwarding and with $P_{raw} = 0.2$, CPI is

$$\begin{aligned}\text{CPI} &= 1 + P_{raw} D_{raw}, \\ \text{CPI} &= 1 + (0.2 \times 0) = 1.0.\end{aligned}$$

There is a substantial improvement in performance with forwarding. If the execution unit had been two stages instead of one, the delay for adjacent instructions with

[64] Note that the memory stage is bypassed even though the time slot is shown occupied. This procedure simplifies the control of the pipeline without hurting its performance.

Petitioner Exhibit 1053, Page 240 of 334

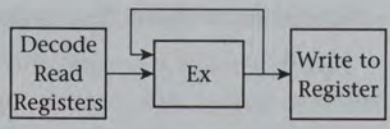
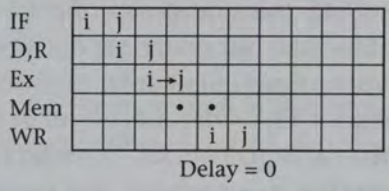


Figure 6.15 Forwarding



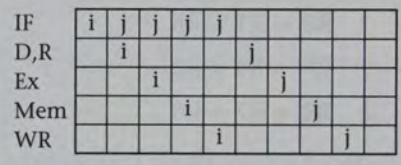
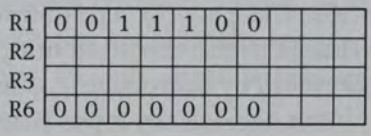


Figure 6.16 RAW control



forwarding would be one clock, and a three-stage execution unit, two clocks. For an execution unit of E stages, the RAW delay for adjacent instructions can be generalized:

RAW delay with forwarding $= E - 1$.

We have discussed the issues of performance in the presence of RAW hazards. In some situations, forwarding cannot be used to reduce delays. How then is the pipeline designed to ensure the logical correctness of the program? In other words, how are the delays inserted when needed? The solution to this problem is found in the use of control bits in the register file as shown in Figure 6.16. The program fragment is the same as that used above:

i R1 = R2 + R3,

j R4 = R1 + R6.

Each word of the register file has a control bit as shown for R1, R2, R3, and R6. When an instruction is decoded, the control bit of the destination register (R1 for instruction i) is set to a 1 on the next clock. A subsequent instruction, such as instruction j, tests the control bits for its source registers, R1 and R6 for this example. If either of the bits are 1, the instruction stalls, the bits are tested on each subsequent clock, and when both become 0, instruction j moves to the next stage of the pipeline. As instruction i writes the result into the register file, the control bit is set to zero, indicating that the RAW dependency has been satisfied. This form of control is used in the IBM S360 integer processor and a number of contemporary microprocessors.

Petitioner Exhibit 1053, Page 241 of 334

Write after Read Dependencies

This dependency occurs when instruction execution is out of order and a later instruction writes into a source operand location before the previous instruction has read that source operand from that location. With the base pipeline of Figure 6.1, the register reads must occur before a subsequent instruction can write into the register. How then can results get out of order? With some processors, the decoder can release instructions out of order to the execution unit(s). For example, if instruction i is waiting for a RAW dependency to be resolved, the decoder can release instruction j, which has no dependencies. Consider the following three-instruction program fragment executing on a processor that can issue instructions out of order:

i R2 = R1 + R8,
j R9 = R4 × R2,
k R4 = R7 + R8.

There is a RAW dependency on R2 between instructions i and j and instruction j stalls. There are no dependencies on source operands of instruction k, which can be released to the execution unit. Instruction k may complete and write into R4 before R4 is read by instruction j, causing an incorrect result to be placed in R9.

This dependency is prevented by buffering the source operands. With instruction i in the decoder, the value in R4 is placed in buffer B1 and the value in R2 is placed in buffer B2 when it is available from instruction i:

j R9 = R4 × R2,
(R4) → B1,
(R2) → B2 when it is available from instruction i.

Instruction j becomes

j R9 = B1 × B2.

The buffering of the inputs to instruction j removes the WAR dependency because instruction k can write into R4 whenever it completes. In many designs, buffers are placed at the inputs to the ALU, and these buffers can be used to resolve WAR hazards. This use of buffers, in their general form, is called renaming.

Write after Write Dependencies

This dependency, also known as an output dependency, occurs when instruction execution is out of order and a later instruction writes into a destination location before the previous instruction has written into that location. Consider the following program fragment:

i R4 = R3 + R1,
j R4 = R7 + R8,
k R9 = R4 + R5.

Petitioner Exhibit 1053, Page 242 of 334

Assume that there is a dependency that stalls instruction i and it completes after instruction j. The completion sequence is j, i, and k:

j R4 = R7 + R8,
i R4 = R3 + R1,
k R9 = R4 + R5.

Instruction k has an incorrect source operand in R4 and the execution is incorrect. Buffering can also be used to resolve WAW hazards. When buffering is used, the program fragment above is rewritten[65] as

i R4 = buff1 = R3 + R1,
j R4 = buff2 = R7 + R8,
k R9 = buff2 + R5.

Even if instruction i completes after instruction j, instruction k will have the correct source operands. However, there is now a RAW dependency on buff 2, which must be resolved as discussed above. With buffers, the dependency is not totally resolved because R4 is still written to the register file out of order. For this reason buffers are not a universal solution to this design problem.

Dependencies in Other Spaces

The preceding paragraphs have discussed dependencies in the general-purpose register file of a processor. Hazards and dependencies of all three types can occur in other spaces as well, including memory, caches, condition code registers, and address registers. The designer must search out and identify all potential dependencies in a design and make sure that there are hardware provisions to ensure that dependencies will not affect the logic of a program.

Buffers will not work for memory-mapped I/O (described in Chapter 7) because the sequence of writes to memory must be in order for the I/O device to operate properly. WAW dependencies can be resolved by forcing the writes to be in order. The PowerPC has an instruction *eieio* (enforce in-order execution of I/O) that forces loads and stores to be in order. This instruction is inserted as a preamble to instructions that will load or store from memory-mapped I/O. The Intel Pentium Pro, which uses memory-mapped I/O, requires that the I/O region of memory be direct and not cached. Memory has hardware that enforces in-order writes, a topic discussed further in Section 7.1.

6.4 INTERRUPTS

The general issues of interrupts, as found in serial execution model processors, are discussed in Chapter 3. A pipelined processor with a number of instructions in simultaneous execution presents a new set of problems to the designer. We like to have precise interrupts. That is, even with a number of instructions in simultaneous execution, the

[65] Rewriting is accomplished automatically by hardware for processors with renaming registers.

Petitioner Exhibit 1053, Page 243 of 334

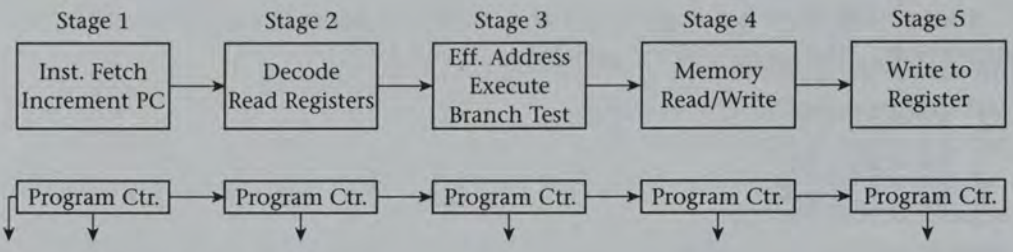


Figure 6.17 Saved program counter

processor can correctly return to the interrupted program and resume execution. If this return cannot be guaranteed, interrupts are said to be imprecise.

When an interrupt occurs, a pipeline stage may be the source of the interrupt, such as an illegal op-code detected in the decode stage. This stage is called the interrupting stage. With an external interrupt, on the other hand, the designer must identify a stage that is the interrupted stage. For the interrupted program to be restarted, the interrupted or interrupting instruction must have its program counter saved.

Any stage of the pipeline can be interrupted or interrupting, depending on the type of interrupt. It follows then that there must be a way of saving the program counter for each instruction that is being processed in the pipeline. This means that there is a program counter pipeline in parallel with the main pipeline, as shown in Figure 6.17.[66]

An important design issue with interrupts is to identify the saved program counter. To address this issue, the three types of interrupts are considered.

Asynchronous or external	These interrupts can occur at any time because of events such as an I/O request. The designer selects the stage of the processor that will have its program counter saved.
Synchronous or internal	These interrupts result from some event in a particular stage of the pipeline. Examples are a page fault, an illegal op-code, or an arithmetic error. The stage in which the event occurs, or the stage −1, has its program counter saved.
Software or programmed	These events are explicit instructions to the processor to perform some function. Examples are a programmed context switch or a call for I/O. The saved program counter is usually the decode stage unless an effective address must be calculated to complete the operation.

[66] There must also be a parallel pipeline to carry control information that tells each pipeline stage what to do.

Petitioner Exhibit 1053, Page 244 of 334

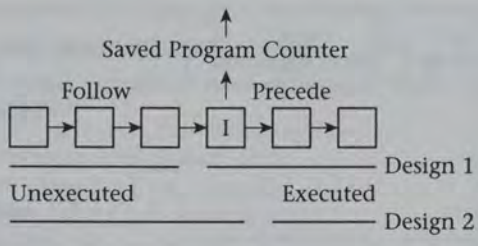


Figure 6.18 Precise interrupt design options (courtesy Jones and Bartlett Publishers)

To have precise interrupts with a pipelined processor, three conditions must be satisfied (Smith and Pleszlumn 1985):

1. Instructions that precede the saved program counter must be completed and have properly modified the processor state.
2. Instructions that follow the saved program counter must be unexecuted and have not modified the processor state.
3. The instruction indicated by the saved program counter must meet either condition 1 or condition 2.

How are the three conditions for a precise interrupt satisfied? Figure 6.18 shows the two design options, design 1 and design 2. Note that different interrupts may be implemented with different designs; design uniformity is not required or achieved in most processors. For example, an illegal op-code detected at the decode stage will have its program counter value saved and it will not be executed (design 2). A breakpoint instruction, detected at the decode stage, will have its program counter value saved and will be executed (design 1).

The unexecuted instructions following the interrupted instruction can be treated in a number of ways. If there are early side effects, such as the autoincrement addressing mode of the VAX architecture, these side effects must be (1) undone and the instructions reexecuted after the interrupt is handled or (2) stored and the instructions restarted after the interrupt is handled. For processors with no early side effects, the following instructions may be abandoned and reexecuted after the interrupt handler is run. RISC design theory requires that there be no early side effects, thus permitting this simple approach to handling following instructions.

With most processors, the preceding instructions are allowed to run to completion. In other words, the preceding instructions are flushed from the pipeline. This strategy works well as long as one of the instructions being flushed does not generate an interrupt. But what happens if a flushed instruction produces an interrupt? The designer must accommodate this event; in some processors, an error signal is produced, indicating a situation that the processor cannot handle.

Other problems include making provisions for two or more simultaneous interrupts, such as an illegal op-code on the same clock as an external interrupt. Again, the designer must consider all such possibilities and provide a solution or provide an error signal to the system.

Petitioner Exhibit 1053, Page 245 of 334

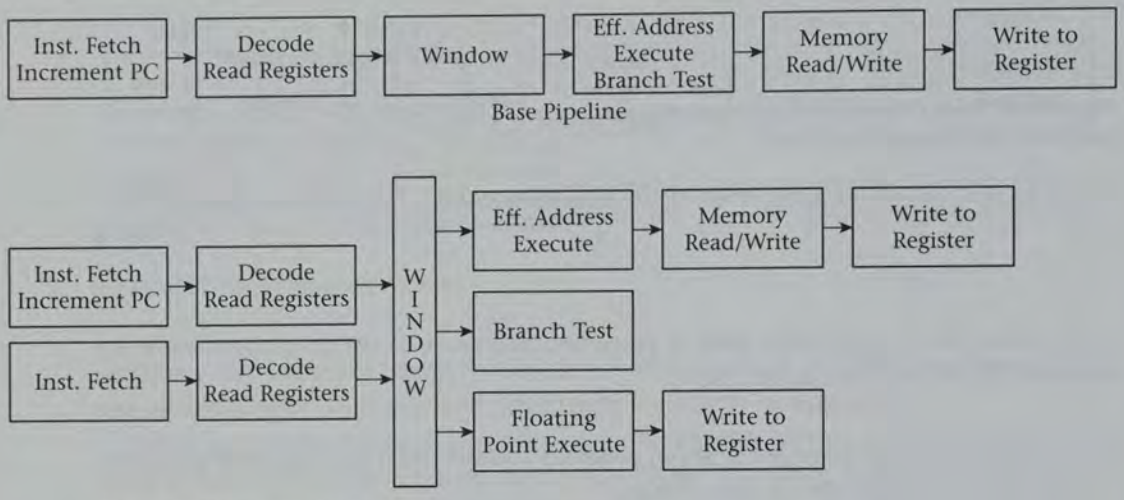


Figure 6.19 Superscalar topology

6.5 SUPERSCALAR PIPELINES

In the preceding sections, the pipelines have decoded and issued one instruction at a time to the execution unit. This results in the steady-state CPI of 1. One may ask, "Why not decode and issue more than one instruction at a time and reduce the steady-state CPI to less than 1?" This is exactly what is done in implementations called superscalar. The idea of superscalar is illustrated in Figure 6.19. The base pipeline is shown at the top of the figure with a window inserted between the decode and the execute stages. For the base pipeline, this window is simply the staging register at the output of the decode stage.

A degree 2 superscalar architecture is shown at the bottom of Figure 6.19. There are two identical pipeline stages fetching and decoding instructions; only one stage increments the PC. The number of replications before the window, also called a dispatch unit, is defined as σ, the degree of superscalar. The figure shows an integer execution unit, a branch execution unit, and a floating-point execution unit. The window releases zero, one, two, or three instructions on each clock. However, the average issue rate cannot exceed two instructions per clock. Because the pipeline stages following the window are not simply replications of the base pipeline, this design is called nonuniform superscalar. Examples of contemporary nonuniform superscalar processors are listed in Table 6.8.

We now derive a performance model for superscalar processors, as illustrated by the reservation tables shown in Figure 6.20. The pipeline illustrated has four stages: instruction fetch, decode and register read, execute, and write back, and each of these stages is duplicated.[67] For this example, $s = 4$, $\sigma = 2$, and $k = 4$.

Because instructions may not be aligned in the instruction cache, two alignment cases must be considered. For case a), the instructions are aligned and the number of clocks required for executing the program fragment is $s + (k/\sigma) - 1$. If the first

[67] This is a form of uniform superscalar that is not found in practice. However, it does serve as a model for performing modeling.

Petitioner Exhibit 1053, Page 246 of 334

TABLE 6.8 EXAMPLES OF SUPERSCALAR TOPOLOGIES

Processor	σ	Execution Units
RS/6000	4	Integer, Flt. Pt. (Mpy, Add), Branch, I/O
PowerPC 601	3	Integer, Flt. Pt. (Mpy, Add, Div), Branch
Intel Pentium	2	2-Integer, Flt. Pt. Add, Flt. Pt. Mpy., Branch
Intel i860	2	Integer-Branch, Flt. Pt. Add, Flt. Pt. Mpy.
AMD-K5	4	Integer, Integer/shifter, Flt. Pt., Branch, Load/Store
MIPS-T5	4	2-Integer, Flt. Pt. Add, Flt. Pt. Mpy/Div/Sqrt, Load/Store
MC88110	2	2-Integer-Branch, Integer Mpy, Integer Div., Flt. Pt. Add, Load/Store, Bit Field, 2 Graphics

two instructions are not aligned, case b), the number of clocks is $s + (k/\sigma)$. With the probability of alignment being $1/\sigma$, the weighted average number of clocks is

$$= \frac{1}{\sigma}\left(s + \frac{k}{\sigma} - 1\right) + \left(1 - \frac{1}{\sigma}\right)\left(s + \frac{k}{\sigma}\right)$$

$$= s + \frac{k-1}{\sigma}.$$

The CPI of a degree $\sigma = 2$ superscalar processor is

$$\text{CPI} = \frac{s + \frac{k-1}{\sigma}}{k} = \frac{1}{\sigma} + \frac{1}{k}\left(s - \frac{1}{\sigma}\right).$$

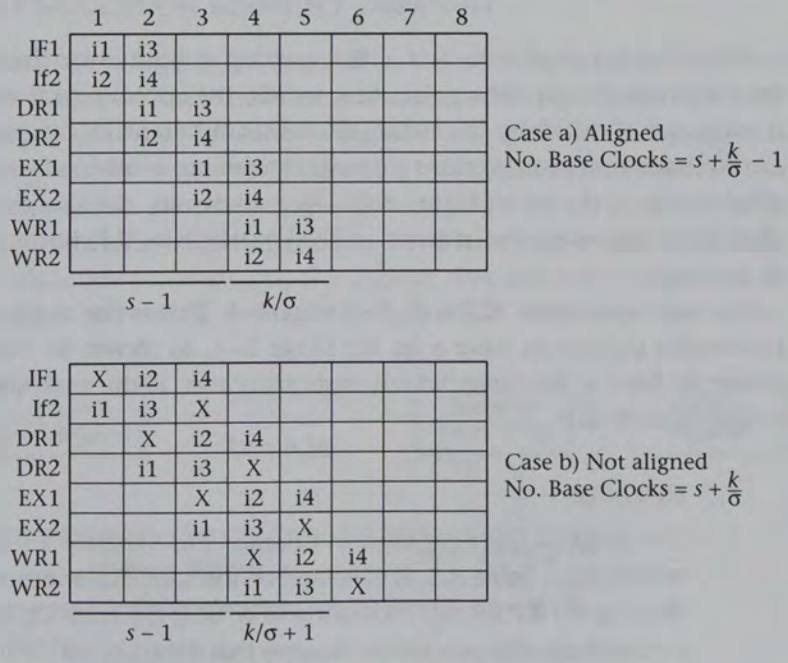


Figure 6.20 Superscalar performance model

Petitioner Exhibit 1053, Page 247 of 334

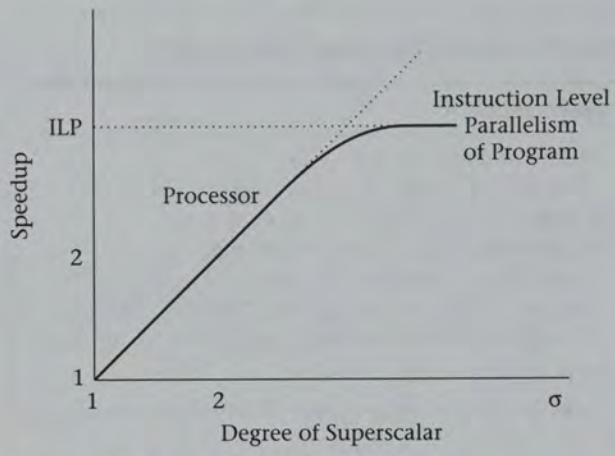


Figure 6.21 Speedup limited by σ and ILP

By setting σ to 1, we see that this is the same model developed above for a nonsuperscalar, $\sigma = 1$, processor. The speedup of a superscalar processor over a nonsuperscalar processor is

$$\text{speedup} = \frac{\text{clocks, nonsuperscalar}}{\text{clocks, superscalar}} = \frac{s+k-1}{s+\frac{k-1}{\sigma}}$$

$$= \frac{\sigma(s+k-1)}{\sigma s+k-1}.$$

As $k \to \infty$, speedup $\to \sigma$.

As $\sigma \to \infty$, speedup $\to 1 + \dfrac{k-1}{s}$.

Note that for large values of σ the speedup is limited by the delays set by k and the length of the pipeline s. In other words, the speedup of a superscalar processor is ultimately limited by the local parallelism inherent in the program. A processor cannot issue more instructions in parallel than are available from the program. This relationship is shown in Figure 6.21. As σ increases, the speedup increases linearly until the point where the instruction level parallelism (ILP) limits any further increase in speedup.

For many problems, ILP is in the range 2–4. This is the reason that most modern superscalar processors have σ in the range 2–4, as shown in Table 6.8. There is no reason to have σ too large, which costs chip area, if the anticipated programs have relatively small ILP.

EXAMPLE

The program fragment used in Chapter 3 to illustrate a load/store register file architecture, Table 3.3, is executed on the superscalar processor of Figure 6.19. What is the ILP for this program and what is the speedup of the processor over a nonsuperscalar processor? Assume that there are no cache misses.

Petitioner Exhibit 1053, Page 248 of 334

Solution

The architecture is three-address, register to register or load/store. The program fragment required for performing the function is:

Load: R(1) ← M(B)
Load: R(2) ← M(C)
Add: R(3) ← R(1) + R(2)
Store: M(A)← R(3)

The superscalar processor uses the integer pipeline for loads and stores as well as the ALU operations – a potential structural hazard. With two-integer pipelines, two load instructions can be issued together; however, the store instruction must follow the add instruction because of the RAW dependency. We can conclude that, without stalls or delays, three clocks are required for this four-instruction fragment. The minimum CPI is $3/4 = 0.75$; however, the CPI may be degraded because of the time required for resolving the RAW dependencies and the structural conflicts in the memory system.

Comment

The RAW dependencies on R1 and R2 are called load true dependencies. These dependencies usually result in a stall delay, even with forwarding.

Note from Table 6.8 that the AMD-K5, MIPS-T5, and the MC88110 have a load/store pipeline in addition to the integer pipeline. This means that for some programs, a load or store with another instruction can be released from the window in one clock.

6.6 PIPELINED PROCESSOR MEMORY DEMAND

A pipelined processor places a significant demand on the memory system for instruction and data fetches. Because of the concurrency in the pipeline, one instruction fetch and up to one data read/write may be initiated in one clock, as shown in Figure 6.22. For superscalar processors, this maximum demand is multiplied by σ. To meet these demands on memory, modern processors have split caches, wide instruction caches for instruction fetching, and write buffers to isolate the data writes from the data reads.

In addition, for RISC-style architectures, the register files must be multiported to support concurrent reads and writes, as shown in Figure 6.23. For dyadic operations,

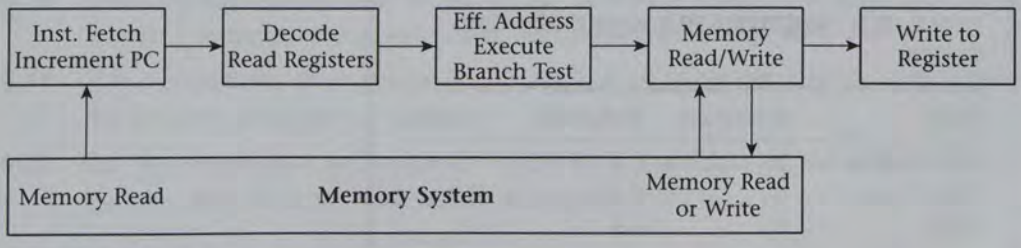


Figure 6.22 Memory demand

Petitioner Exhibit 1053, Page 249 of 334

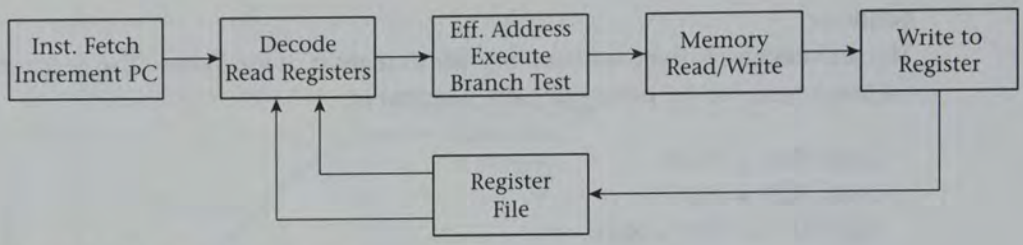


Figure 6.23 Register file accesses

there are two reads and one write per instruction. For superscalar processors the ports to the register file must, for the worst case, support σ(2 reads + 1 write) register cycles per clock.

EXAMPLE

Consider a superscalar processor, $\sigma = 2$, and workload. 30% of all instructions are ALU types, 30% are loads or stores, and the other 40% of instructions do not use the register file. What is the weighted average number of register file accesses per clock?

Solution

An event probability table is constructed in Table 6.9. The weighted average number of register accesses is 1.2σ and, for $\sigma = 2$, the weighted average number of register file accesses per clock is 2.4.

Comment

The value of 2.4 register file accesses per clock is a worst-case value. Because of pipelining, the accesses will not occur simultaneously as suggested by this solution. Simulations confirm this by showing that approximately 90% of all clocks will have three or fewer register file references for a $\sigma = 4$ processor (Johnson 1991).

A designer may be tempted to replicate the register file rather than make it multiported as a way of solving the structural hazard. If this is done, the design must ensure that the contents of the register files are coherent, a design problem that may cost more chip area and be slower than the multiport design because of the need to maintain coherency between the register files.

TABLE 6.9 REGISTER FILE ACCESSES PER CLOCK

Event	References	Probability	Product
ALU instruction	3σ	0.3	0.9σ
Load or store	σ	0.3	0.3σ
Other	0	0.4	0
		SUM = 1	1.2σ

Petitioner Exhibit 1053, Page 250 of 334

REFERENCES

Cragon, H. G. (1991). *Branch Strategy Taxonomy and Performance Models*, IEEE Computer Society Press, Los Alamitos, CA.

Davidson, E. S. (1971). "The Design and Control of Pipelined Function Generators," presented at the IEEE Conference on Systems, Networks and Computers, Oaxtepec, Mexico, January 1971.

Johnson, M. (1991). *Superscalar Microprocessor Design*, Prentice-Hall, Englewood Cliffs, NJ.

Kogge, P. M. (1981). *The Architecture of Pipelined Computers*, Hemisphere, New York.

Lee, R. L. (1984). "Branch Prediction Strategies and Branch Target Buffer Design," *Computer*, **21**:1.

Smith, J. E. and Pleszlumn, A. R. (1985). "Implementation of Precise Interrupts in Pipelined Processors," *IEEE Transactions on Computers*, **C-34**:3, pp. 36–44.

EXERCISES

6.1 For modeling a superscalar processor, how many alignment cases must be considered for $\sigma = 3$, 4, or 5?

6.2 Show that the CPI model developed for $\sigma = 2$ from Figure 6.20 is also the general solution for any σ.

6.3 Show with reservation tables that the RAW delays with forwarding are equal to $E - 1$.

6.4 What is the RAW delay for various values of E and spacing between instructions with the dependencies? Show in table form.

6.5 Derive the WABD model for the delayed branch strategy.

6.6 Show all the steps in the derivation of the optimum number of pipeline stages.

6.7 Consider an L2 unified cache that receives misses from instruction and data L1 caches. Assume that the number of references that are correlated is the product of the two misses: P_{miss} (instruction) $\times$ P_{miss} (data). Assume an ISA that has one instruction and one data reference per instruction. If the miss rate for both L1 caches is 0.1, what is the mean access time of the unified cache?

6.8 Show by the use of reservation tables that the forwarding delay is $E - 1$ clocks for adjacent instructions.

6.9 Describe a situation in which the write to register stage of the pipeline can cause an interrupt and be the interrupting stage.

6.10 Show the derivation of the model for the number or clocks, $s + [(k - 1)/\sigma]$, for a degree 2 superscalar processor with the probability of alignment $1/\sigma$.

6.11 What conditions or situations could result in no instructions being issued from the window of Figure 6.19?

6.12 For the superscalar processor of Figure 6.19, consider the two-instruction sequence: load floating point, add floating point

$$i\,\mathrm{R}x = \mathrm{M}(\mathrm{R}y)$$
$$j\,\mathrm{R}z = \mathrm{R}x + \mathrm{R}w.$$

Petitioner Exhibit 1053, Page 251 of 334

Show a forwarding path that will improve the performance of this program fragment. Explain how performance is improved.

6.13 Identify all forwarding paths on the superscalar processor of Figure 6.19 that will improve performance loss that is due to RAW dependencies.

6.14 The base pipeline has $s'' = 2$, $s' = s = 3$. Compute the CPI for the three branch strategies, freeze, predict take, and predict not take. The workload parameters are $P_b = 0.4$ and $P_{bt} = 0.6$.

6.15 Rework the example in Subsection 6.3.2 to include cache misses. The one-level unifi ed cache is organized direct with 8K bytes. With the split cache, the instruction and data caches are direct with 4K bytes. Use P_{miss} data from Tables 4.8–4.10. The real memory to the unifi ed cache transport time is 10 clocks. See Figure 4.10 for the topologies of the one-level unifi ed cache and the one-level split cache.

6.16 You are redesigning the BTB of a processor and can increase the number of bits by 50%. How would you spend these additional bits? Increase the number of entries, thereby decreasing the miss rate? Increase the size of the history table to improve the accuracy of prediction? Any other ideas? Justify your answer based on the reduction in WABD.

6.17 Rework the WABD model for the BTB, Table 6.7. Change the default case when there is a BTB miss to be predict take and predict not take.

6.18 Rework the example in the Section on BTBs by using the PT and PNT Strategies as default. Which default strategy would you use for this design? Explain your answer.

6.19 Examine the weighted average number of register file accesses per clock by passing the following instruction stream down the base pipeline: $\sigma = 1$. The instruction stream is load, ALU, load, other, ALU, other, other, other, ALU, store. Compute the fraction of instructions having 0, 1, 2, and 3 register file references. How do these results compare with the result of Table 6.3 and the reported simulation?

6.20 Split caches, instruction and data, remove the structural hazard from memory references. Discuss the requirements for maintaining coherency between these two caches.

Petitioner Exhibit 1053, Page 252 of 334

SEVEN

INPUT/OUTPUT

7.0 INTRODUCTION

Processors need a source of input data and a destination for output data. Processors also require storage space for large volumes of intermediate data. Thus the I/O system provides these storage facilities by connecting the processor's memory to the I/O devices. These devices vary widely in latency and bandwidth; thus, a major function of the I/O system is to match their latencies and bandwidths to that of the memory system. I/O devices consist of such diverse items as keyboards, modems, and hard disks. Each of these devices has its own latency and bandwidth requirements that must be satisfied for a properly functioning system.

Blaauw and Brooks (1997) classify I/O devices with the taxonomy shown in modified form in Figure 7.1. Store devices provide ready-access storage space. System resident and database devices are usually in the virtual-address space of the memory, providing low-latency storage space. Archive stores, on the other hand, usually require human intervention when archival information is required or needs to be stored.

Source/sink devices are used as paths to/from the computer to another device and are not generally used for storage of information. Floppy disks and Zip disks can be viewed as either archive or local media. Windows98 can use the Internet Explorer as an access method for both store devices and source/sink devices, blurring the distinction of the taxonomy.

7.1 I/O SYSTEM ARCHITECTURE

The I/O system must provide the facilities for moving information between the processor's memory and the various devices. The nature of the interface leads to major design problems. There can be a significant bandwidth difference between the memory and the device – as high as 5,000,000:1 and as low as 1:1. For example, memory may have a bandwidth of 50M bytes/s whereas a keyboard may produce input at approximately 10 bytes/s. In addition, a graphics display may accept data at 30M bytes/s. There can also be a significant latency difference – as high as 500,000:1 and as low as 1:1. The latency difference between memory (70 ns) and a disk (20 ms) is an example of these latency differences. The differences in bandwidth and latency lead to the design

Petitioner Exhibit 1053, Page 253 of 334

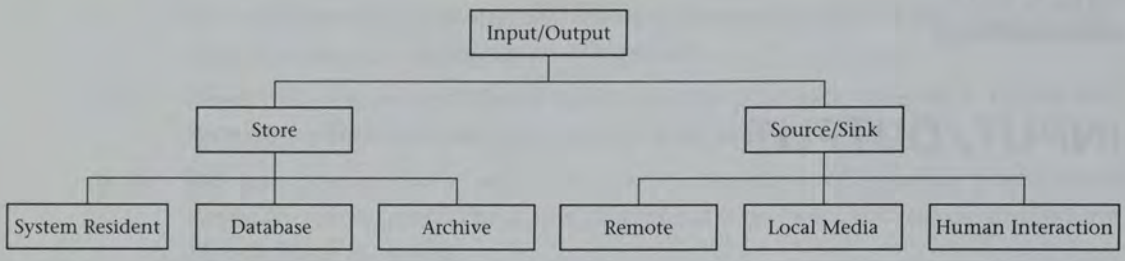


Figure 7.1 I/O device taxonomy

requirement for buffers. These buffers can be implemented in hardware or virtulized in real memory with software.

The I/O devices are usually asynchronous to the clock of the processor and memory, and also to the program executing in the processor. Asynchronism requires synchronizers at the hardware level, discussed in Section 7.5, and synchronization primitives in the processor ISA, discussed in Chapter 8.

The major tasks of the I/O system are the following:

1. to establish the connection between the memory and the I/O device
2. to synchronize and manage the control of the data transfer
3. to provide buffering when the source and sink bandwidths and latencies differ
4. to perform code conversions if required
5. to interrupt the processor when required
6. to terminate the operation when it is completed

These tasks are accomplished by hardware and software, in varying amounts of each. The more hardware oriented, the higher the performance and the higher the hardware cost. The more software oriented, the lower the performance and the lower the hardware cost. Control can be classified into two broad categories: programmed I/O and coprocessor I/O.

Programmed I/O

Programmed I/O, also known as direct I/O, is accomplished by a program executing on the processor itself to control the I/O operations and transfer the data. With programmed I/O, shown in Figure 7.2, there are several architected registers (minimally an input register and an output register) that are addressed by special move instructions.

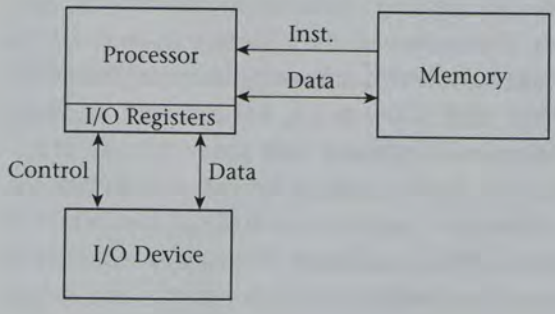


Figure 7.2 Programmed I/O

Petitioner Exhibit 1053, Page 254 of 334

These move instructions move information from the accumulator (or general-purpose register) to/from the I/O registers. The peripherals are connected directly to the bits of these registers.

The register bits are allocated for control signals, to and from the device, and for the data path. Some implementations have a special status register to hold status and control bits so that control is not mixed with data in the same register. There may also be a counter register for counting the number of transfers so the processor will not be required to count under program control.

When the main program requires an I/O operation, the main program is suspended and an I/O subroutine is called. This subroutine performs the bit-by-bit control and data transfer between the memory and the device. After the I/O transfer is completed, control is returned from the I/O subroutine to the suspended main program. In some situations, the I/O program is inserted as an in-line code as an open subroutine.

The most significant problem with programmed I/O is that concurrency between I/O and computing is not possible. The main program is suspended while the I/O subroutine is executed, leading to performance loss.

EXAMPLE

Consider a processor with programmed I/O that executes instructions at a rate of 0.1 MIPS. An I/O subroutine for writing to a floppy disk requires 10,000 instructions and the disk has a total latency of 100 ms. A disk request is initiated every second of task time. What is the effi ciency of this system?

Solution

The time that the main program is suspended is 100 ms + I/O subroutine time:

$$\begin{aligned}\text{suspend time} &= 100 \times 10^{-3} + \frac{10{,}000}{100{,}000}\ \text{s} \\ &= 100 \times 10^{-3} + 100 \times 10^{-3}\ \text{s} \\ &= 200 \times 10^{-3}\ \text{s},\end{aligned}$$

$$\begin{aligned}\text{efficiency} &= 100\left(\frac{\text{task time}}{\text{total time}}\right) \\ &= 100\left(\frac{1\ \text{s}}{1\ \text{s} + 200\ \text{ms}}\right) \\ &= 83\%.\end{aligned}$$

Comment

Today, many single-chip microcomputers, or controllers, use programmed I/O. The loss of efficiency is not a concern as the major design issue is very low cost. When the I/O control subroutine is placed in very dense ROM, the total chip area is reduced from the area required by other control techniques.

Programmed I/O was used on the first computer built on the von Neumann ISA model, the EDSAC at Cambridge University built under the direction of Maurice Wilkes (Randell 1973). Recall from Chapter 3 that the EDSAC had the ability to call and return from subroutines. Punched paper tape was the input medium, and the output was a

Petitioner Exhibit 1053, Page 255 of 334

teleprinter. Input and output commands were provided in the ISA:

I, *n* Read the next row of holes on the input tape and place the resulting integer, multiplied by 2^{-16}, in the storage location *n*.

O, *n* Print the character set up on the teleprinter, and set up the next character represented by the five most-significant digits in storage location *n*.[68]

An early production computer that used programmed I/O is the IBM 701, which was first delivered in 1953. The 701 had four magnetic drums for store; for source and sink it had a card reader, card punch, printer, magnetic tape, and a graphic display with a camera for producing hard copy.

The 701 has I/O instructions that are at a higher lexical level than the EDSAC. Two instructions, read select RDS and write select WRS, established the path to an addressed device and started the device operating (such as starting a tape drive). Other instructions performed the read or write operation. All transfers were by means of the multiplier quotient (MQ) register:

CPY Copy and skip	RDB Read backwards
WEF Write end of file	SNS Sense
REW Rewind tape	LDA Locate drum address

Simple input IN and output OUT instructions were used on the Intel 8080. These instructions used the memory bus pins and moved a byte to/from the accumulator and the data bus pins along with a byte address on the memory-address pins. Various higher levels of I/O, such as buffers and channels, were implemented on special chips. Three status bits were present on pins that were externally decoded to determine if an I/O operation instead of a memory operation was taking place: 101 = I/O write, 110 = I/O read.

Single-chip microcontrollers such as the Intel i8048, shown in Figure 7.3, are commonly used for the control of simple devices such as microwave ovens, keyboard controllers, and handheld games. These computers use a programmed I/O that moves either bits or bytes.

The Intel 8048 has three 8-bit I/O ports and an 8-bit bus. The ISA contains a number of instructions that perform input and output moves between the ports and the accumulator. Outputs can be masked by use of immediate operands for bit-setting purposes.

For upward compatibility reasons, these In and Out instructions are still found on the Pentium Pro. With this processor, the source/destination can be any of the registers AL, AX, or EAX, depending on the size of the addressed port (8, 16, or 32 bits). Windows95 uses programmed I/O for some of its I/O functions.

Memory-Mapped I/O

Memory-mapped I/O is another form of programmed I/O that maps the device connections to bits in memory address space, as shown in Figure 7.4. An output command is a normal store instruction that stores a pattern of bits into a designated memory location. An input command is a normal load instruction that reads a memory location.

[68] Note the CISC-like function of the output instruction: print one character, set up the second character.

Petitioner Exhibit 1053, Page 256 of 334

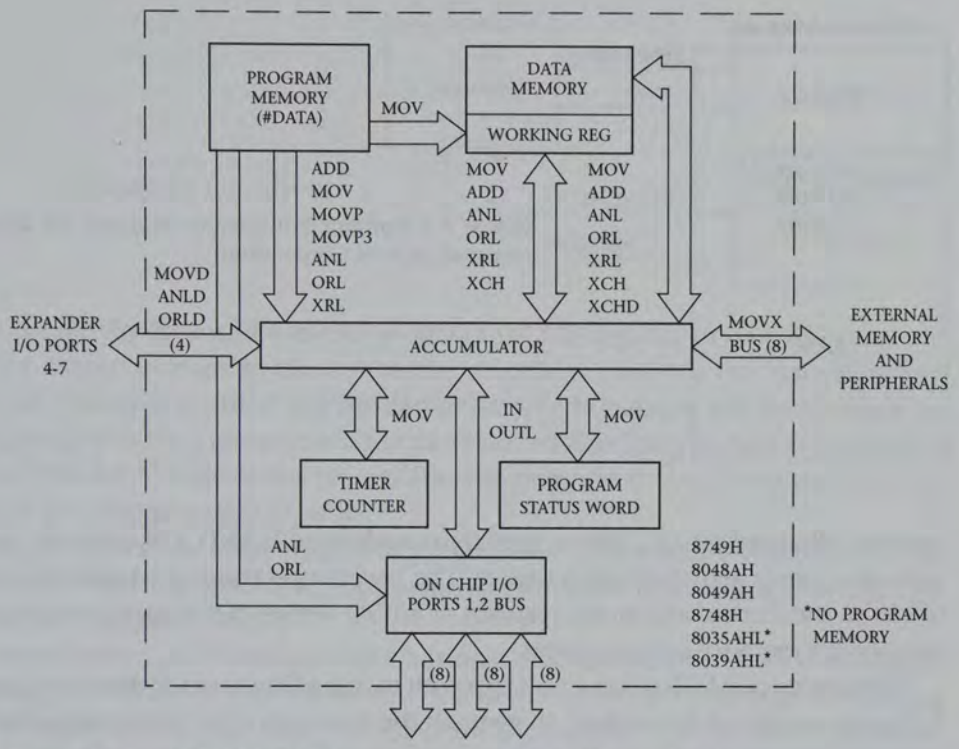


Figure 7.3 Intel 8048 block diagram (courtesy of Intel Corporation)

The major advantage of memory-mapped I/O compared with programmed I/O is that no special move instructions or special I/O registers are required. The major disadvantage is that a portion of address space must be treated specially by both the hardware and software designers. For the hardware designers, the memory-mapped bits must be real flip flops and their memory addresses blocked out of the memory-address decoder. From a software point of view, any design that interferes with a uniform memory allocation presents a design problem.

Memory-mapped I/O requires the full participation of the processor, as with programmed I/O. Thus there is a similar loss of efficiency that is due to executing the I/O control program.

The Intel Pentium Pro supports memory-mapped I/O in addition to programmed I/O. Any of the processor's instructions that reference memory can address memory-mapped I/O locations. An example of an allocation is shown in Figure 7.5. The move instruction can transfer data between any of the processor's registers and the spaces in

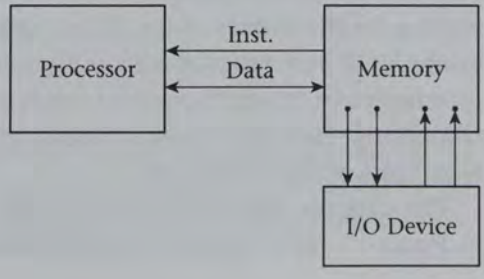


Figure 7.4 Memory-mapped I/O

Petitioner Exhibit 1053, Page 257 of 334

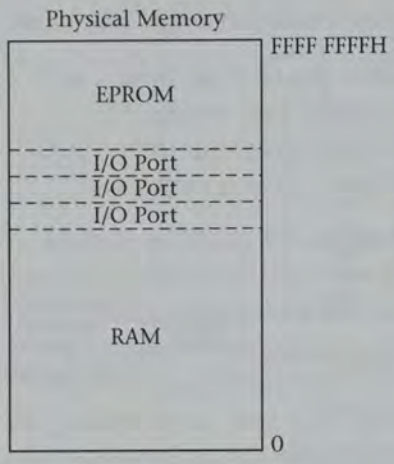


Figure 7.5 Pentium Pro memory-mapped I/O allocation (courtesy of Intel Corporation)

memory allocated for I/O. Other operations such as add, AND, OR, and test can also have one source argument in I/O space. The logical operations are useful for control of bits in the control and status registers of an I/O device that is mapped into one or more of the memory-address spaces.

Memory-mapped I/O poses a problem with caching. How are the devices connected if the I/O spaces can be cached? In general, the answer is that I/O is mapped into an uncached region of memory. This space is uncached for two reasons: cache page misses will make the time for I/O nondeterministic and fast memory reads and writes are not as important for I/O as for processor instructions and data.

In addition, the Pentium Pro does not cache a memory-mapped address so that writes are in order, a topic discussed in Subsection 6.3.3. If two instructions controlling a peripheral device by memory-mapped I/O complete out of order, the sequence of information sent to the device is incorrect and improper operation will result. For example, the command bits could be presented to the device after the data bits.

The noncached addresses are established with a set of memory-type range registers (MTRRs) that set the range limits for memory spaces. By setting limits, the MTRR registers map I/O spaces into uncached addresses.

As the IBM S360 does not use memory-mapped I/O, out-of-order writes are not an I/O problem. In-order writes are required for other purposes, and provisions are made to ensure in-order writes. The PowerPC *eieio* instruction, discussed in Subsection 6.3.3, ensures in-order writes to memory.

Interrupt I/O

In the efficiency example above, we saw that 50% of the idle time was devoted to waiting for the latency of the floppy disk. Because of this low efficiency, there are few examples of pure programmed or memory-mapped I/O in high-performance systems. Interrupts can be used to recover much of the idle time, making this time available for computing. Early mainframe computers, such as the Univac 1103 and the IBM 701, were users of this technique.

The data connection between the I/O device and the processor memory is shown in Figure 7.6 as a memory-mapped system. The processor sends a command to the

Petitioner Exhibit 1053, Page 258 of 334

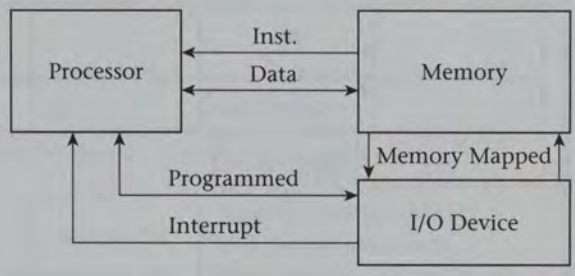


Figure 7.6 Interrupt I/O

device, usually by means of a special programmed I/O register, to start its operation. After the request is made to the device, the processor returns to the main program. When the latency is satisfied and the device is ready to transfer, the device sends an interrupt signal to the processor and the processor services the interrupt and performs the data transfer. Various status reports are sent from the device to the processor by means of the programmed I/O paths.

Interrupt I/O can be used with both programmed and memory-mapped I/O; the Intel Pentium Pro implements both techniques. The details of handling interrupts are discussed in Chapters 5 and 6.

Note that interrupt I/O will provide additional processing only if there is additional processing to be done or if the request for I/O can be made far in advance of the time it will actually be required. One or both of these conditions can be met with multiprogrammed systems, discussed in Chapter 4, but not with single-user workstations without background processing.

EXAMPLE

Assume that interrupt I/O is added to the programmed I/O example. Initiating the disk transfer requires 100 instructions, and responding to the interrupt at the end of the disk latency requires another 100 instructions. What is the efficiency of the system?

Solution

The 100-ms latency can be used for useful task time work. However, the total time is increased by the time required for executing the 200 instructions associated with the subroutine call and return:

$$\begin{aligned}\text{useful time} &= 1 + 0.1 \text{ s} \\ &= 1.1 \text{ s}, \\ \text{total time} &= 1.2 + \frac{200}{100{,}000} \text{ s} \\ &= 1.202 \text{ s}, \\ \text{efficiency} &= 100\left(\frac{1.1}{1.202}\right) = 91.5\%.\end{aligned}$$

Comment

We see that the use of interrupts improves the efficiency to 91.5% from 83%. However, it is possible in some situations that the instruction time added for the interrupt can be greater than the time saved by overlapping processing with the device latency.

Petitioner Exhibit 1053, Page 259 of 334

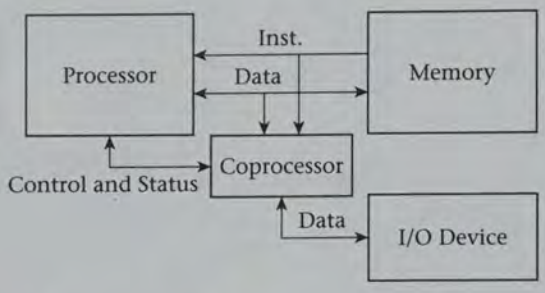


Figure 7.7 Coprocessor I/O

Coprocessor I/O

When the processor is being used to control I/O and move the data, there is a loss of processor efficiency. Thus the idea naturally arises to use a second processor or coprocessor for I/O. The autonomous coprocessor first appeared in the IBM 709 in 1959, and it performed all the I/O tasks and freed the main processor for its tasks. Figure 7.7 shows a coprocessor connecting an I/O device to the system bus. Control and status information is passed between the processor and the coprocessor by means of paths similar to those of programmed I/O. The processor sends a command to the coprocessor that autonomously performs the I/O operation.

The introduction of a coprocessor makes this configuration a multiprocessor. The problems of any multiprocessor must be addressed with coprocessor I/O. These problems include software synchronization, hardware synchronization, memory protection, and memory coherency. For students interested in these problems and their solutions, a course in multiprocessors is recommended.

There are two general types of these coprocessors: programmed I/O channel and direct memory access (DMA).

Programmed I/O Channel. A programmed I/O channel is an additional processor that is added to the system to provide I/O services by means of the programmed I/O technique. This processor, with a special-purpose ISA, is fully autonomous and can operate in parallel with the main processor.

The source of instructions for the programmed I/O channel can be either the main memory, shown in Figure 7.7, or a private memory. The main memory design is found in the IBM 360/370/390 channel. Channel control programs are placed in memory by the operating system. When the processor needs I/O service, a pointer is loaded into the channel's program counter and execution is started. The IBM channel operation is discussed further in Section 8.4.

The private memory design is found in the CDC 6600 peripheral processor. Each of the peripheral processors has its own local program memory to hold its control program. The advantage of a private memory is that main memory instruction bandwidth is not consumed during an I/O operation.

Comparing the ISAs of the IBM S360/370/390 channel with the CDC 6600 peripheral processor, one finds that the ISA of the IBM channel is high level and very CISC-like. On the other hand, the ISA of the CDC peripheral processor is low level and very RISC-like.

Direct Memory Access (DMA). DMA is a variation of programmable I/O. The DMA coprocessor receives a list of parameters from the operating system that describes the

Petitioner Exhibit 1053, Page 260 of 334

TABLE 7.1 CONTROL METHODS

Type of I/O	Advantage	Disadvantage
Programmed	Simple, little extra hardware	Loss of processor efficiency to execute I/O subroutine. Requires special registers and move instructions
Memory mapped	Simple, little extra hardware. No special move instructions	Loss of processor efficiency to execute I/O subroutine. Requires special address logic for memory addressing
Interrupt; programmed or memory mapped	Simple, little extra hardware, improved efficiency by the processor using device latency times	Programmed: requires special registers and move commands. Memory mapped: requires special address logic for memory addressing
Coprocessor; programmed channel or DMA	Fully concurrent, no loss of processor efficiency. Transfers are direct to/from memory bypassing the processor registers	Requires significant additional hardware. May require that main memory bandwidth be increased to support extra instructions and data transfers

nature of the transfer. Examples of these parameters are type or transfer, device name, starting memory address, length of the transfer, and various control parameters. The parameter list can be transferred to the DMA by means of the system bus or a dedicated path similar to programmed I/O paths.

The DMA control is usually implemented as a hardwired state machine. The parameter list provides the inputs to the state machine; it controls and performs the transfer as specified by the parameters.

Summary of I/O Control Methods

Table 7.1 shows a summary of the I/O control methods discussed above. The advantages and disadvantages are noted.

7.2 I/O DEVICE REQUIREMENTS

The application of the processor determines the type and the quantity of the I/O devices. These requirements range from almost nothing for dedicated controllers to vast amounts for large mainframe installations. We consider two processor applications: personal computers and mainframes (commonly called servers today).

Personal Computer Devices

PCs are used in a single-user environment with limited multiprocessing activity. In general, low-latency devices are favored when the volume of data moved is small whereas high-bandwidth devices are favored where the volume of data moved is large. Operating systems such as Windows95 and Windows98 support and use virtual

Petitioner Exhibit 1053, Page 261 of 334

TABLE 7.2 PC I/O DEVICES

Store	System resident	Hard disk	4–17G bytes 5400 RPM 8.5-9.5 ms average seek latency
	Database	CDROM	650M bytes per disk 1.8 –16M bytes per second 75–118-ms average latency
	Archive	Zip drive	100M bytes per disk 29-ms average seek time 20M bytes per second 3-s average start/stop time
Source/sink	Remote	Modem	56K baud V.90 standard
	Local media	Scanner	Flatbed Single pass Line art, halftone, gray scale, and color 75 dpi, 150 dpi, 300 dpi, 600 dpi
	Human interaction	Keyboard	104 key Data rate: typing rate
		Monitor	17-in. SVGA 0.27-mm dot pitch 1280 × 1024 noninterlaced resolution 30–70-kHz horizontal synchronization 50–100-kHz vertical synchronization

memory. Relatively large hard disk space is needed to keep a large number of large programs resident. Interconnection to various communications systems is a requirement with the widespread use of the Internet. Archival storage is needed to archive data and back up the system.

Large-screen, high-quality video is needed to support graphical programming languages and for graphics applications such as drafting, CAD, and games. High-quality audio is needed for entertainment systems and CDROM devices that support local databases.

To meet these requirements, PC manufacturers provide a wide variety of I/O devices for their customers. A small list of these devices and their characteristics are given in Table 7.2 with the taxonomy from Figure 7.1.

Mainframe or Server Devices

Mainframe and servers provide computing resources to a number of users by means of terminals and periodic processing of large batch jobs. Many mainframe installations primarily provide database facilities with only a modest requirement for computing. Typical of mainframe applications are airline reservations, credit card verification, and order entry. These applications are known as transaction processing systems; a transaction request queries a database, conducts a transaction, and updates the database to reflect the transaction. Transaction processing systems are usually real time, that is, a customer is waiting until the transaction can be completed. Thus the performance

Petitioner Exhibit 1053, Page 262 of 334

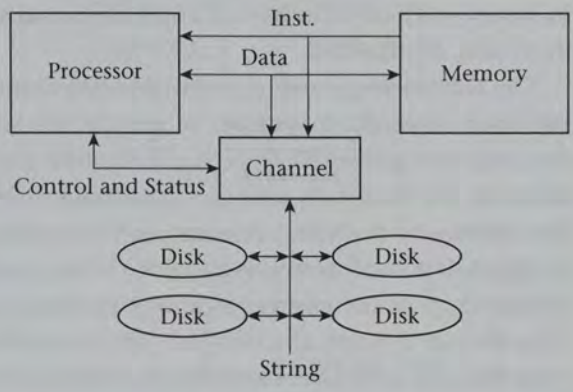


Figure 7.8 Mainframe disk string

of these systems is usually measured in transactions per second. During the Christmas season, a credit card company's server may need to respond to 50,000 transaction requests per second.

The major I/O device requirements for mainframe are usually of two types: hard disk for system resident store and network access from remote terminals. Studies by IBM have shown that the hard disk requirement is approximately 4.5G bytes per MIPS of computing capacity.[69] For a 500-MIPS mainframe, 2250G bytes of disk are needed.

EXAMPLE

A transaction processing system is considered for a PC. The PC executes instructions at 100 MIPS and has a disk with a total mean access time of 20 ms. Four-disk accesses and 100,000 instructions are needed for each transaction. How many transactions per second are possible with this system?

Solution

The total time for a transaction is the disk time plus the processor time:

$$\text{transaction time} = 4 \times 20 \times 10^{-3} + \frac{100{,}000}{100 \times 10^{6}}$$

$$= 80 \times 10^{-3} + 1 \times 10^{-3} = 81 \text{ ms},$$

$$\text{transactions per second} = \frac{1}{81 \times 10^{-3}} = 12.35.$$

Comments

Even if the processing time goes to zero, the transaction rate is limited by the four-disk accesses. For PCs to be effective transaction processors, the I/O system must be improved to support many concurrent disk accesses. This is exactly what is being done by a number of PC manufacturers today.

The one-disk unit usually found with a PC does not have sufficient capacity, bandwidth, and low latency to meet the needs of transaction processing. Thus a number of disk units must be attached to an I/O channel, as shown in Figure 7.8. A channel is attached to the disks by a path called a string. The strings are usually redundant so that operation can continue in the event of a string failure. In addition, a large mainframe

[69] A typical PC may have 0.005G bytes per MIPS.

Petitioner Exhibit 1053, Page 263 of 334

or server will have a number of controllers and a number of strings as several hundred disks may be attached.

The remote access requirements for mainframes are met with various dedicated and switched networking systems. In general the volume of data transferred for transaction processing is small; thus the bandwidth requirement is modest. Communication between the terminals and the mainframe is by land lines, microwave, or satellite. The latency of dedicated communications paths, even by means of geosynchronous satellites, is a small component of the total latency time because of disk accesses. Key components of the communications system are the switching computers that route information through the network. At the mainframe, the communications lines are connected to a channel controller that queues requests for mainframe service.

The World Wide Web is different from transaction processors in that both low latency and high bandwidth are required. Low latency is needed for response time and the high bandwidth is needed because of the large volume of graphics, video, and sound that is being transmitted. The WWW will be discussed further in Subsection 7.6.1.

7.3 BUSES AND CONTROLLERS

The first two I/O tasks noted in Section 7.1 concern the establishment of connections between the memory and the I/O devices and the control of data transfers over this connection. Most modern computers use a bus, or buses, for interconnecting the major components of the system. A bus serves as a pathway between components for passing addresses, instructions, and data. A simple high-level view of a bus is shown in Figure 7.9. This is the bus shown in Figure 7.6 that is used with the coprocessor I/O. This bus interconnects the processor, main memory, and I/O devices by means of their controllers. Also recall the discussion in Chapter 4 on the role played by the bus in transferring instructions and data between the processor and memory.

Buses can be classified in a number of ways: by purpose, control, and communication technique (Thurber et al. 1972). The options for bus design are as follows:

Purpose: dedicated or general purpose
Control: centralized or decentralized
Communications: synchronous or asynchronous

From the list above, we see there are eight possible design points for a bus. In the following sections, the characteristics of each of the six design parameters are discussed and correlated to bus designs used in computer systems today.

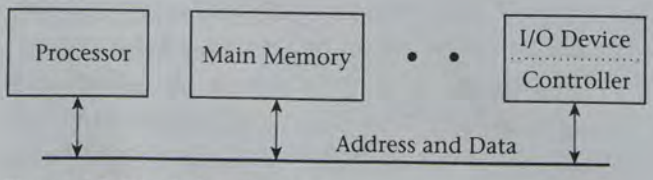


Figure 7.9 Simple bus structure

Petitioner Exhibit 1053, Page 264 of 334

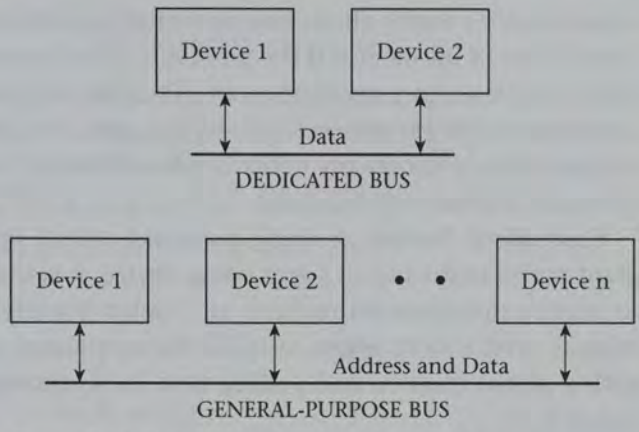


Figure 7.10 Dedicated and general-purpose buses

Dedicated or General Purpose

The major difference between dedicated and general-purpose buses is that a dedicated bus is point to point between two physical devices, whereas a general-purpose bus interconnects more than two physical devices. Figure 7.10 illustrates the two types. Dedicated buses are used in cases in which the latency and the bandwidth requirements are such that sharing the bus with another user can result in unacceptable system performance. In these cases, a special dedicated bus is provided by the designer. Note that a dedicated bus does not need address lines because the source and destination are addressed by implication. That is, device 1 always sends to device 2 or device 2 to device 1. Some dedicated buses are unidirectional with information flow in only one direction. For example, the bus that connects a memory to a graphics controller may be unidirectional.

Because dedicated buses are used internal to the processor or for special high-bandwidth applications without general-purpose capabilities, these buses are not considered further. Instead, the following paragraphs discuss the control and communications design techniques found with general-purpose buses.

General-purpose buses differ from the processor internal buses discussed in Chapter 5. A dedicated internal bus is controlled by an outside state machine or microprogrammed controller, in contrast to a general-purpose bus in which the address of the receiver is provided by the sender.

Also, with a general-purpose bus, a number of users share the same bus and simultaneous requests for the bus are resolved by one of a number of resolution techniques. Some of the devices on a general-purpose bus are both senders and receivers or only senders or only receivers. For example, a printer controller's primary function is to receive data and send some status information. A disk controller, on the other hand, sends and receives data and sends status information.

Centralized or Decentralized Control

The control of a general-purpose bus can be either centralized or decentralized. The basic requirement is to grant or not grant a device access to the bus. With centralized or

Petitioner Exhibit 1053, Page 265 of 334

decentralized control, all devices are treated equally except for their priority of access. Thus, if one of the devices is the processor, it may be given the highest priority for bus access. However, in some systems an I/O device may have the highest priority because of the loss in system performance; for example, if a disk is unable to read and misses a complete revolution, many milliseconds will be lost. A thorough discussion of priority is outside the scope of this book.

Centralized Control. A single hardware control unit will recognize a request and grant access to the bus to a requesting device. It is the responsibility of the controller to resolve simultaneous requests and assign priority to the requests. At least three designs, with modification, are used for centralized controllers: daisy chain, polling with a global counter, and polling with local counters. These designs are shown in Figure 7.11.

Convention calls the unit holding the bus controller the master whereas the control logic in the device is called a slave. Note that there are many variations on the basic designs discussed below.

DAISY CHAIN. With this design, the control lines to the devices are connected in series. When a device needs access to the bus, it sends a bus request to the bus controller. The controller then sends a bus-available query to the first device to determine its current status of activity. If the device does not want access to the bus, it passes the bus-available query on to the next device, and so on. The requesting device responds with bus-busy and is given the next access to the bus.

The advantage of daisy chain control is its simplicity, low cost, and expandability. The disadvantages are reliability; if a line breaks, all of the devices downstream are removed from the system. Also, this control favors upstream device requests leading to "hogging" of the I/O system by the upstream devices.

POLLING WITH GLOBAL COUNTER. Each device can issue a bus request to the controller. When this request is received, the controller polls the bus-request lines to identify the requester and to find which device, if any, is using the bus. The polling sequence is established by a counter in the controller that starts at 1 and increments for each poll. When the counter value equals the device number, the requesting device is identified. If the bus is idle, or when it becomes idle, the request is granted.

This system is more reliable than daisy chaining but has the same priority-favoring characteristic because the counter starts at 1 for each poll cycle. However, we can eliminate this disadvantage by assigning random addresses to the devices or by using a pseudorandom global counter. Expansion, within the limits of the controller design capacity, is relatively easy.

INDEPENDENT REQUESTS. Central control with independent requests receives bus-request signals from the various devices that can be viewed as interrupts to a processor. The status of the bus is maintained in the controller; all the grants of bus access are determined based on this status information. When the controller determines that a device can use the bus, a bus-granted signal is sent to the device.

This system is flexible because the priority can be determined by programming the bus controller with a table of priorities. Latency time is smaller than with the other two systems discussed because there is reduced communication between the controller and the devices.

Petitioner Exhibit 1053, Page 266 of 334

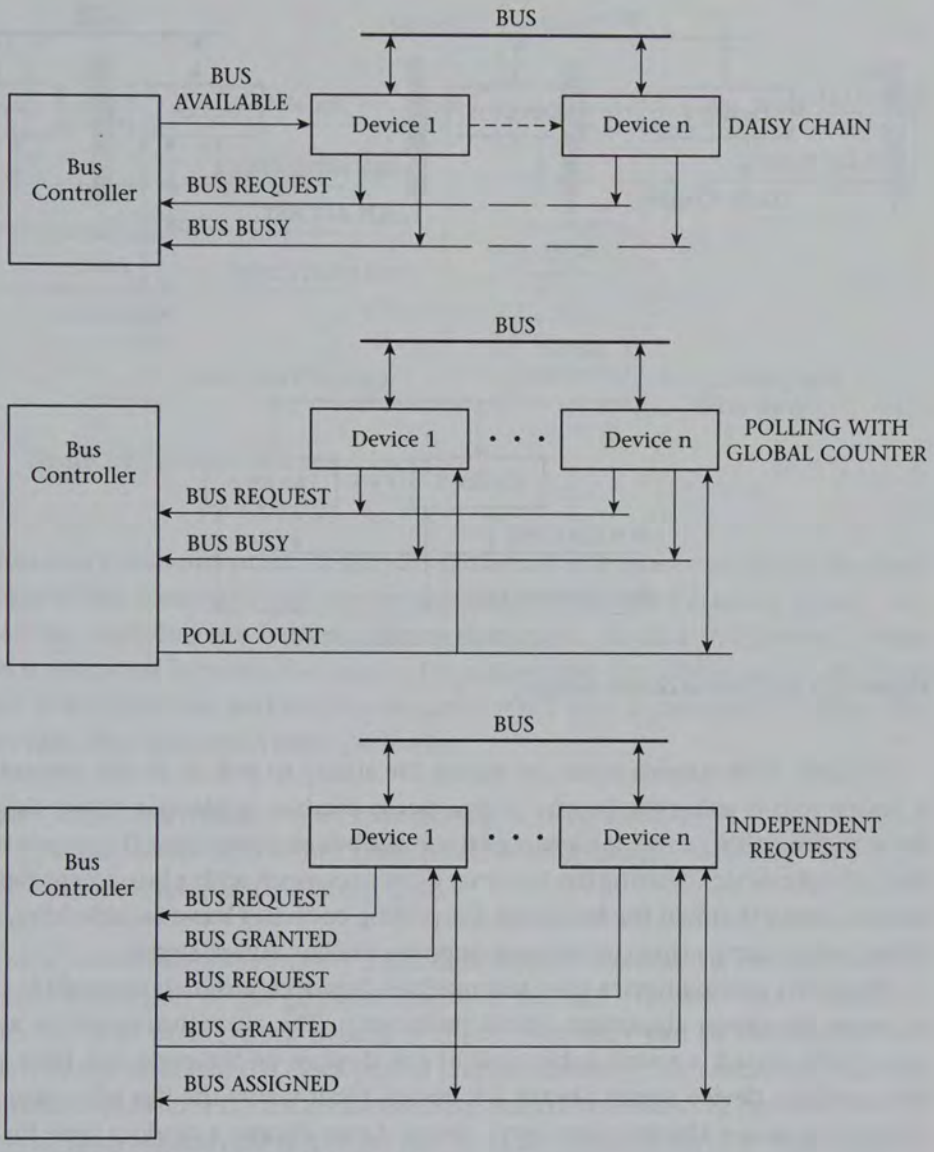


Figure 7.11 Centralized control designs

Distributed Control. Distributed control, also known as decentralized control, distributes the control function between all the devices on the bus. The major advantage of decentralized control is that the system is easily expandable by the addition of modules. As with centralized control, there are three basic designs with variations: daisy chain, polling, and independent requests. Figure 7.12 shows these designs in simplified form.

DAISY CHAIN. A device issues a bus request to the bus-request line that is attached to bus available. When a device receives the bus available, if it is not the originator of the request, it passes it on to the next devices. This continues until the requesting device receives the bus-available signal, at which time the request is granted. If the bus is being used, none of the devices can issue a bus request.

Petitioner Exhibit 1053, Page 267 of 334

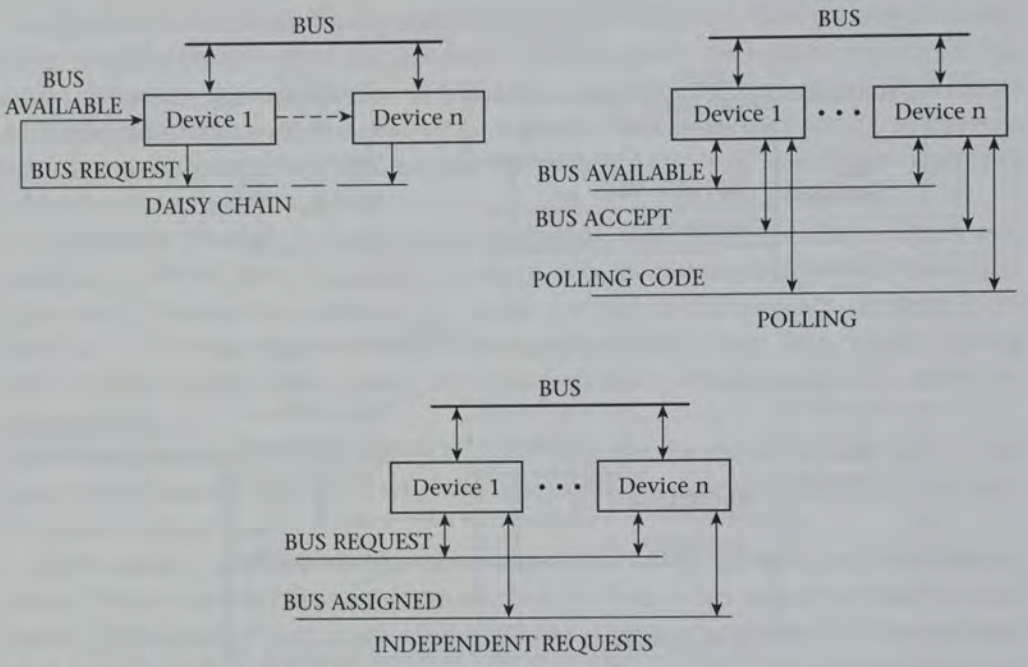


Figure 7.12 Distributed control designs

POLLING. This scheme works by giving the ability to poll to all the devices. When a device that is using the bus no longer needs the bus, it places a device-dependent code on the polling code line and signals on the bus-available line. If the code matches the code of a device desiring the bus, that device responds with a bus-accept signal. The former device that had the bus drops the polling code and bus-available lines, and the latter device starts using the bus and removes the bus-accept signal.

When the polling device does not receive a bus-accept signal, the code is changed by some allocation algorithm, and it polls again. The algorithm could be as simple as a count giving a round robin poll of the devices or the code can have priority. For example, device 3 may always ask device 7 if it wants the bus after device 3 has completed its use. On the other hand, device 4 may choose a random code for its poll.

INDEPENDENT REQUESTS. With this scheme, when any device is using the bus, it signals on the bus-assigned line. Another device desiring to use the bus places a request on the bus-request lines consisting of a code for its priority. When the current user finishes with the bus and releases it, it also removes its signal from the bus-assigned line. All devices examine all the bus requests, and the device that recognizes itself as having the highest priority takes control of the bus and signals on the bus-assigned line.

Synchronous or Asynchronous Communication

The transmission of addresses, control information, and data between two devices may be synchronous with a clock, or asynchronous without a clock and self-timed (Del Corso et al. 1986, and Di Giacomo 1990). There are advantages and disadvantages to each, as discussed in the following paragraphs.

Petitioner Exhibit 1053, Page 268 of 334

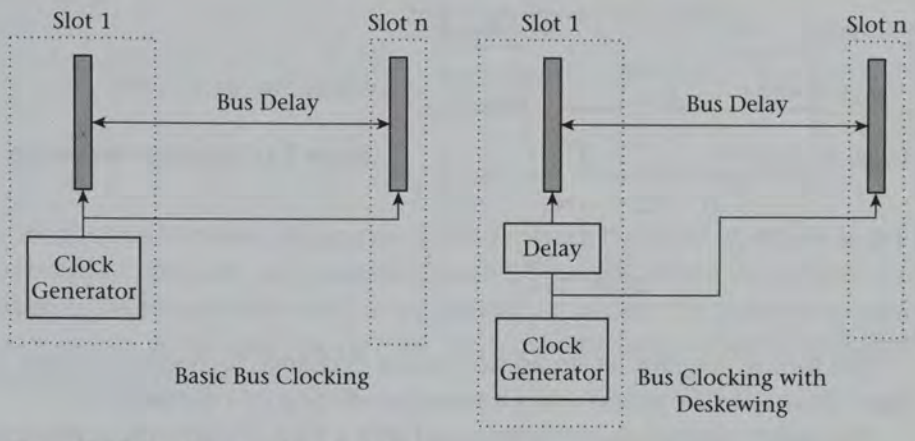


Figure 7.13 Synchronous bus clocking

Synchronous Communication. A simplified diagram of a synchronous bus (the data and clock portion) connected between two devices is shown on the left of Figure 7.13. Note that this logic diagram looks similar to that of the pipeline in Figure 6.5. Data are to be transmitted between the card in the right-hand slot to the card in the left-hand slot. The transmitter and the receiver are clocked from a common source on the left-hand card. The minimum clock period is

$$\begin{aligned} t_{clk}(\text{min}) &= \text{bus delay(max)} + [\text{clock skew(max)} + \text{setup time}] \\ &= \text{bus delay(max)} + [\text{Bus Delay(max)} + \text{setup time}] \\ &\approx 2 \text{ bus delay(max)} + \text{setup time.} \end{aligned}$$

The clock period of approximately 2 bus delays sets a limit on the bus bandwidth. This limit is a function of the physical length of the bus. However, the clock skew represents one half of the clock period. If that 1 bus delay can be eliminated, the bandwidth can be doubled; the clock period is approximately 1 bus delay per transfer.

To eliminate this factor from the clock period, extensive deskewing circuits are used to reduce the skew at each card slot in the bus to an acceptable level of 1–2 gate delays. Delays, shown on the right of Figure 7.13, are inserted in the clock lines to the cards; the delays are the inverse of the distance from the clock generator. Thus the clock arrives at all the cards at the same time. The use of deskewing increases the bus bandwidth by approximately a factor of 2. Even if only one additional bus slot is occupied and it is the one closest to the transmitter, the clock must be set in anticipation that at some time a distant slot will be occupied.

Asynchronous Communication. Asynchronous communication for buses was developed to overcome the worst-case clock rate limitations of synchronous systems. With asynchronous communications, data transfers occur at the fastest rate and smallest delay possible under the circumstances of the physical status of the bus. As cards are added to the bus, the timing automatically adjusts for each card. There are a number of asynchronous protocols; however, only one of the simpler ones is discussed here.

Petitioner Exhibit 1053, Page 269 of 334

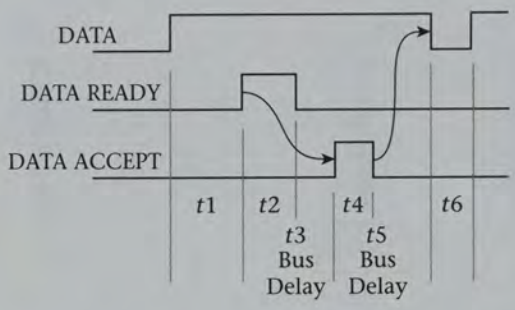


Figure 7.14 Asynchronous transfer

The timing diagram for an asynchronous exchange is shown in Figure 7.14. This figure illustrates the action when a source sends data to a destination.

The source places data on the bus, and after a time $t1$ transmits a data ready signal that has width $t2$. When the destination receives the data ready signal after time $t3$, it latches the data and may make a parity check. The receiver then informs the transmitter with a data accept signal that the data transfer has been successful. If there is a problem with the data, a data error signal (not shown) is sent to the source. The source can initiate another transfer after $t6$.

This process is rather slow because it requires a minimum of two bus delays ($t3+t5$) before a second transmission of data by the source. There are other protocols that reduce the number of delays and increase the bandwidth of the system, but these are outside the scope of this book.

7.3.1 BUS DESIGN EXAMPLES

There are eight design points that use the design criteria discussed above. Dedicated buses are tailored to the needs of a particular transfer path, such as the data path buses shown in Figure 5.4. However, general-purpose buses are normally standardized and are of the greatest interest to us. Table 7.3 provides examples of the four general-purpose bus types from the taxonomy: centralized and decentralized control and synchronous and asynchronous communications.

7.4 SERIAL COMMUNICATION

The buses discussed above are usually designed for parallel transfer of a number of bits, 32 for example. There are cases in which, because of the data rates or the transfer

TABLE 7.3 GENERAL-PURPOSE BUS DESIGN EXAMPLES

Control	Communications	Examples	Specification
Centralized	Synchronous	PCI	PCI SIG v2.1.
Centralized	Asynchronous	IPI	ANSI X3.129
Decentralized	Synchronous	Multibus II	ANSI/IEEE 1296
Decentralized	Asynchronous	VME	IEEE 1014
		FutureBus	IEEE 896.1

Petitioner Exhibit 1053, Page 270 of 334

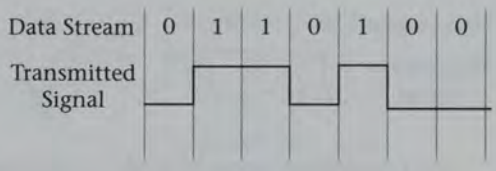


Figure 7.15 NRZ encoding

medium, serial transfer is used. Examples of this type can be found in mouse to processor, keyboard to processor, and communications by means of telephone lines. The first problem to be solved with serial communications is that the transmitter sends clocked data to the receiver and the receiver must, in some way, recover a clock from the transmitted data stream.

Clock Recovery

For most serial communication systems, data transfers are conducted over varying distances, and transmitting a clock with zero skew is impractical. For these cases, the data bit stream must be encoded in such a way as to permit a clock to be recovered by the receiver. These techniques are called implicit clock schemes.

Three techniques for clock recovery are discussed in the following paragraphs. Section 7.5 discusses the problem with synchronization between the receiver's clock and the data for which clock recovery is not possible.

Nonreturn to Zero (NRZ). The NRZ coding transmits one level for a 1 and another for a 0, as shown in Figure 7.15. The receiver's clock, which is set at the same frequency of the transmitter's, is synchronized with each transition of the incoming signal. One of the problems with this coding is that for long runs of ones or zeros, no transition occurs and the receiver's clock is without a synchronization input. For this reason, the stability of the receiver's clock oscillator and the synchronization method are important.

A new standard universal serial bus (USB) is becoming available on PCs for interfacing relatively low-bandwidth devices. This standard uses NRZ with bandwidths of 12M bits/s up to 5 m. A low-speed subchannel at 1M bits/s is supported for interfacing to devices, such as a mouse, for which cost is a consideration.

Frequency Modulation (FM). With the FM technique, a different frequency is transmitted for a 1 and a 0, as illustrated in Figure 7.16. Because there are transitions within each bit period, the clock can be recovered with greater ease.

Some modems use a type of FM that increases the bandwidth of the serial channel. For example, a four-frequency system can encode as follows:

Freq. 1 00
Freq. 2 01
Freq. 3 10
Freq. 4 11

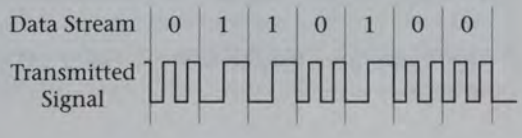


Figure 7.16 Frequency modulation

Petitioner Exhibit 1053, Page 271 of 334

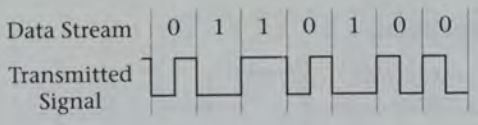


Figure 7.17 Manchester biphase-space code

The transmitter modem looks at the transmitted data stream two bits at a time and selects the frequency to transmit during a fixed period of time. The receiving modem detects the frequency and reconstitutes the data stream.[70]

Manchester Code. Another clock recovery technique is known as the Manchester code, of which there are several variations. Figure 7.17 illustrates the biphase-space version. This code ensures a transition between each bit position, which facilitates recovery of the clock. Note, however, that the code is context dependent, as illustrated by the two consecutive ones. In general, the decoder must examine two bits at a time to determine the actual data stream.

Serial Communications Protocols

Serial communication protocols can be taxonomized by the clocking method and format of the transmitted data.

Clocking	
Synchronous	Clock is distributed or bits are transmitted continuously to provide clock synchronization.
Asynchronous	The line is in a quiescent state, either a 1 or 0, until a message is transmitted, at which time the receiver synchronizes its clock.

Data	
Character	Encoded characters, such as ASCII.
Block	Binary message.

There are four combinations of these two characteristics. Three of these are discussed below: asynchronous character, synchronous character, and synchronous block. There is no known use of an asynchronous-block protocol.

Because these systems transmit a stream of bits, the receiver must first be able to recover the clock from the transmitted data stream, as discussed above. After the recovery of the clock, with a stream of ones and zeros, where does a character or message start and end? Thus it is necessary to provide coding that will permit character and message synchronization and recovery. Morse code provides a simple example: After each of the variable length characters is transmitted in the form of dots and dashes, the transmitter pauses before starting the next character. Word separation is identified by context.

Asynchronous-Character Protocol. With this protocol, the line is quiescent until the transmitter emits a start bit followed by the character bits, as shown in Figure 7.18. Depending on the character set, the number of character bits may be as large as 8 and

[70] The Touch Tone telephone is a form of FM encoding: 10 digits plus 2 control symbols are encoded.

Petitioner Exhibit 1053, Page 272 of 334

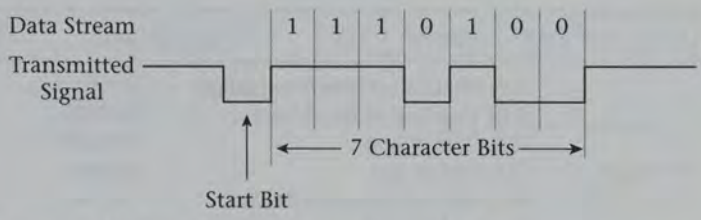


Figure 7.18 Asynchronous-character protocol

a parity or check bit may be added. The character specification must be agreed on by both the transmitter and the receiver. After the character bits are transmitted, the line is returned to the quiescent state. Some period of time must elapse before another character is transmitted.

Another asynchronous-character protocol is the RS232 standard. For this standard, the character type and the clock rate must be initialized by both the sender and the receiver. During the quiescent state the line is at zero, also known as mark, and the start of a character transmission is indicated by a 1, also known as space. During the quiescent time, the receiver samples the input line at twice the bit clock rate to detect the start bit.

Table 7.4 shows the RS232 character codes. There are three character codes: 5-bit code, 7-bit code, and 8-bit code. For the 7- and the 8-bit codes there are programmable variations.

Synchronous-Character Protocol. This protocol surrounds a text message, either fixed or variable length, with various synchronization and control information, as shown in Figure 7.19. The transmitter continuously sends a stream of SYN characters, 0010110, so that the receiver maintains bit and character synchronization. The receiver continuously looks for the bit pattern 0000001 (SOH), which indicates the beginning of a message that establishes synchronization of the message. The header provides such information as the length of the text message. Following the header is the STX character, then the body of the message begins. The message is terminated with the ETX character, which is followed by block character check information.

TABLE 7.4 RS232 CHARACTER CODES

Type	Bit Stream
5-Bit code	Start bit + 5 data bits + 1–2 mark bits
7-Bit code	Start bit + 7 data bits + 1–2 mark bits
7-Bit code	Start bit + 7 data bits + 0 + 1–2 mark bits
7-Bit code	Start bit + 7 data bits + 1 + 1– 2 mark bits
7-Bit code	Start bit + 7 data bits + even parity + 1–2 mark bits
7-Bit code	Start bit + 7 data bits + odd parity + 1–2 mark bits
8-Bit code	Start bit + 8 data bits + 1–2 mark bits
8-Bit code	Start bit + 8 data bits + even parity + 1–2 mark bits
8-Bit code	Start bit + 8 data bits + odd parity + 1–2 mark bits

Petitioner Exhibit 1053, Page 273 of 334

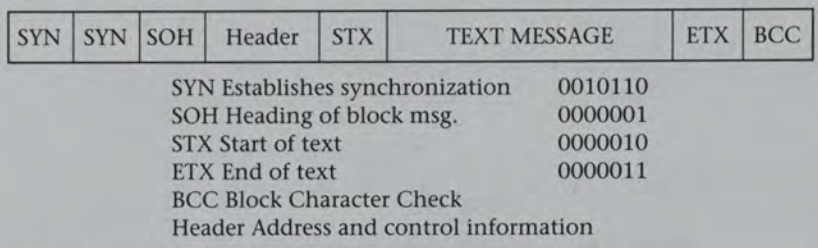


Figure 7.19 Synchronous-character protocol

For this protocol the text message characters are 7-bit ASCII with odd parity. The ASCII character set reserves the control codes that are guaranteed not to appear in a text message.

Synchronous-block protocol: One of the many block oriented protocols for message synchronization is the high-level-data link control (HDLC). This protocol, shown in Figure 7.20, precedes a message with a preamble called a flag. During a quiescent period, flags are continuously transmitted to maintain clock and character synchronization. The start of a message is indicated by the address field that is constrained to exclude 01111110. The message is terminated with a frame check sum to ensure that the data has been received as transmitted (within the ability of the check sum).

The binary message can have any combination of ones and zeros; even the flag bit pattern 01111110 may exist in the message. Thus the transmitter must look for this pattern in the message and must insert a zero, called bit stuffing, to break up this pattern:

Message 01111110
Becomes 011111010

The receiver, having first synchronized on the address and control, removes the inserted zero.

EXAMPLE

With the HDLC protocol, if the message has five ones followed by a zero (111110), how does the receiver know that the 0 was not stuffed and is a legitimate message bit?

Solution

The pattern that is looked for by the receiver is 011111010, a 9-bit pattern. This pattern has the stuffed zero.

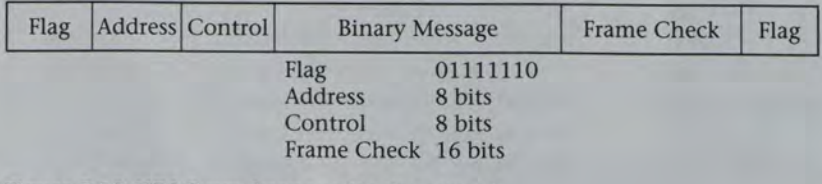


Figure 7.20 HDLC synchronous-block protocol

Petitioner Exhibit 1053, Page 274 of 334

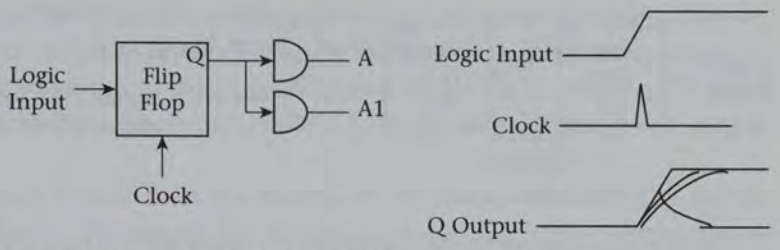


Figure 7.21 The metastability problem

7.5 CLOCK SYNCHRONIZATION

The issue of synchronization between two or more programs will be discussed in Chapter 8. In this section, we discuss the problem of synchronization between two hardware systems that communicate but have separate clocks (Kleeman and Cantoni 1987) and for which clock recovery is not a design option. For example, an I/O channel connects a disk memory (with its clock) to the real memory of the processor (with its clock).

Figure 7.21 illustrates the problem of unsynchronized clocks. A logic input signal from a system with another clock is input to a flip flop with its own clock. The two clocks are not synchronized. If an asynchronous logic input change occurs during the clock pulse or clock transition, the flip flop can be metastable for a period of time, with the final state being nondeterministic. If the output of the flip flop is used to drive two drivers with unequal thresholds, then the driver's outputs may not be logically equal for some period of time. That is, A is not equal to A1. As these signals are used as inputs to other gates, the outcome cannot be predicted and unreliable system behavior can result.

Figure 7.22 shows oscilloscope-sampled Q and Q' outputs of two cross-coupled transistor–transistor logic NAND gates (Chaney and Molnar 1973). The R and S inputs are switched simultaneously to emulate the simultaneous input logic change and the clock. Note that the time for metastability to subside is unbounded. A small fraction of switching events stay in the metastable state for a very long time; longer than 9 ns in the case of Figure 7.22.

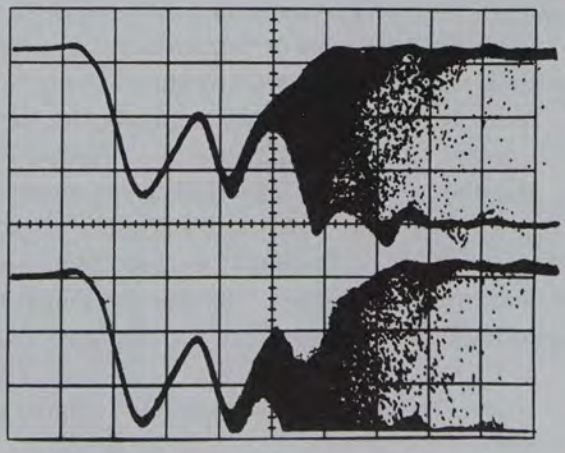


Figure 7.22 Synchronizer metastability, 1 ns/div, 1 v/div (© 1973 IEEE)

Petitioner Exhibit 1053, Page 275 of 334

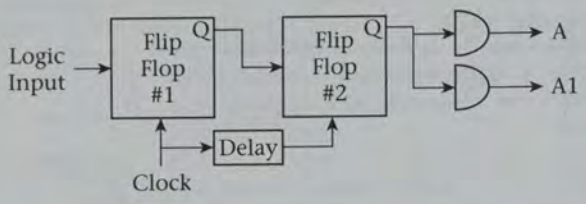


Figure 7.23 Synchronizer with reduced metastability failure rate

The problems of erroneous operation resulting from metastability occur infrequently and are very difficult to diagnose because the errors are transient.

EXAMPLE

To illustrate the magnitude of the synchronization problem assume the following. The system attempts to synchronize every 16 clocks (for a processor memory bus handshake), the clock period is 100 ns, and the synchronizer fails every 1×10^{11} attempts.

Solution

The mean time between failure (MTBF) of synchronization is

$$\begin{aligned} &= \text{number of clocks} \times \text{clock rate} \times \text{synchronizer failure rate} \\ &= 16 \times (100 \times 10^{-9}) \times (1 \times 10^{11}) \\ &= 1.6 \times 10^{5}\ \text{s} \\ &= 1.85\ \text{days}. \end{aligned}$$

Comment

This MTBF is small enough to make the system appear to be unreliable and long enough to make diagnosis of the problem very difficult. [71]

The preceding example suggests the two methods for reducing synchronization MTBF: reduce the frequency of synchronization and/or reduce the MTBF of the synchronizer. Remember that a failure-free synchronizer has been proven impossible to design.

The frequency of synchronization can be reduced by providing that more bits will be transferred on each synchronization. For example, synchronizing for the transfer of a 1K-byte block rather than synchronizing on every bit transferred will reduce the synchronization frequency by a factor of 8192.

After the frequency of synchronization is reduced, the MTBF of the synchronizer may be reduced. There are a number of techniques that have been used. The first step is to use very high-frequency transistors for the synchronizer. This technique reduces the period of time in which the synchronizer may be metastable.

Another technique for reducing synchronization MTBF is shown in Figure 7.23. This synchronizer works as follows. The clock is delayed for a period of time until the probability of flip flop 1 being in the metastable state is reduced to an acceptable value. Note from Figure 7.22 that the longer the wait, the lower the probability of metastability. After this delay, flip flop 2 is clocked. Because of the delay, the input to

[71] A rule of thumb for designers is that the MTBF of a good synchronizer system should be approximately three years.

Petitioner Exhibit 1053, Page 276 of 334

flip flop 2 has a low probability of changing and it may not go metastable. However, because the maximum time that flip flop 1 can be metastable is unbounded, there is a small probability that the input to flip flop 2 is changing during its delayed clock and it too will go metastable. This synchronizer has a finite probability that it will exhibit metastability.

The time delay must be less than the period of the clock; otherwise the system will not function properly. The reason for this is that if another input is presented to flip flop 1 while it is metastable, it will probably remain metastable. Also, the delay must be greater than the likely maximum time of metastability. These are difficult, and contradictory, design requirements to meet for systems with very high-frequency clocks.

EXAMPLE

Take the synchronizer of Figure 7.23. Assume that 15 ns is deemed to be a sufficient clock delay to eliminate most, if not all, metastable conditions on flip flop 1. What is the minimum clock period and maximum clock rate of this system?

Solution

$$\text{minimum clock period} = 15 \text{ ns},$$

$$\text{maximum clock rate} = \frac{1}{15 \text{ ns}} = 66.6 \text{ MHz}.$$

7.6 QUEUING THEORY

When there are a number of requests for the same resource, called a structural hazard in Chapter 6, many systems establish queues to hold the requests until they can be serviced. Queuing theory was developed to permit closed-model evaluation of these systems without resorting to computer simulation. The two models discussed below are quite simple and ignore many of the complexities of some systems. Nevertheless, these models permit rapid and fairly accurate evaluation of these systems. The most important result from these models is the mean or average time required for servicing a request. The two models discussed are open systems and closed systems. A designer can usually identify the type of system being considered and apply the proper model.

7.6.1 OPEN-SYSTEM QUEUING MODEL

The open system is shown in Figure 7.24. The queue is first in first out (FIFO). Requests for service arrive, are placed in the queue, and are served when the request reaches the

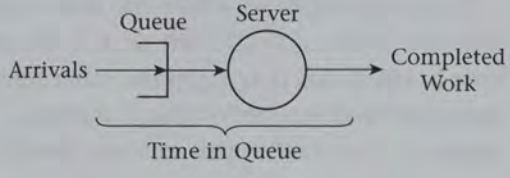


Figure 7.24 Open-system queuing model

Petitioner Exhibit 1053, Page 277 of 334

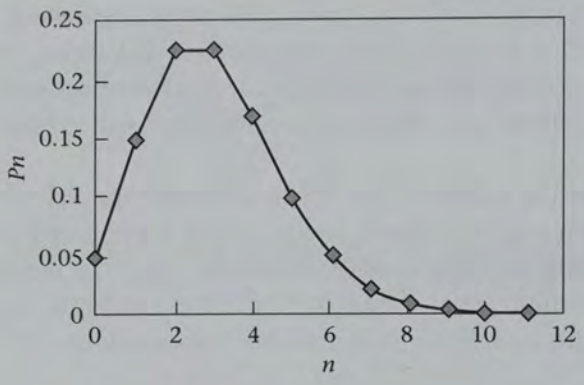


Figure 7.25 Poisson distribution, $\lambda = 3$

tail of the queue. The time in the queue is the total time in the system (queue time plus server time).

The open queuing model is applicable not only to disk I/O systems but to airport runways, grocery checkout counters, and Internet servers. The derivation of the model for the time in the queue is based on two assumptions:

1. The arrival rate and the service rate follow a Poisson distribution.
2. Requests arrive, are served, and never request service again, within the time frame of the model.

The Poisson distribution

$$P_n = \frac{\lambda^n}{n!} e^{-\lambda},$$

for $\lambda = 3$, is shown in Figure 7.25.

We now define some terms. The mean arrival rate is λ, the mean service rate is μ, and the mean time in the queue (queue time + service time) is t_q.The model for t_q, without derivation, is

$$t_q = \frac{1}{\mu - \lambda}.$$

Consider the limits of λ:

As $\lambda \to 0$, $t_q \to 1/\mu$,

and

as $\lambda \to \mu$,$t_q \to \infty$.

From experience we know that if we walk up to a bank teller, for example, and there is no one in line (in other words, $\lambda \to 0$), we will be served immediately and our mean time in the queue is $1/\mu$. On the other hand, when customers are arriving at the same mean rate that service is being provided ($\lambda \to \mu$), the mean time in the queue for some customers may be infinite.

Petitioner Exhibit 1053, Page 278 of 334

EXAMPLE

An Internet server can perform 100 transactions per second, and requests are arriving at the rate of 80 requests per second. What is the mean time in the queue?

Solution

$$t_q = \frac{1}{\mu - \lambda} = \frac{1}{100 - 80} = \frac{1}{20} = 0.05 \text{ s.}$$

Comments

The mean time in the queue is 0.05 seconds, five times that of the unloaded system; 0.01 s.

It is not necessary to know the values of λ and μ. The model can be normalized by use of the duty cycle, also known as utilization, of the server, a parameter easier to estimate or measure in some situations. The duty cycle is defined as $\rho = (\lambda/\mu)$, and

$$t_q = \frac{1}{\mu - \lambda} = \frac{\frac{1}{\mu}}{\frac{\mu}{\mu} - \frac{\lambda}{\mu}} = \frac{1}{\mu(1 - \rho)}.$$

The normalized time in the queue is plotted in Figure 7.26 for $\mu = 1$. Note that for a duty cycle of 0.5, the time in the queue is twice the service time. For a duty cycle of 0.8, the time in the queue is five times the service time – the identical results of the previous example. The plot stops at a duty cycle of 0.9 because the asymptotic behavior of the model gives an infinite time in the queue for a duty cycle of one.

Figure 7.27 shows the system response time versus system utilization for an array of magnetic disks (Chen 1989). The value of t_q is approximately 35 ms at a duty cycle of zero. Further, at $\rho = 0.5$ the response time is approximately 70 ms, as determined by the model. The model is reasonably accurate up to a duty cycle of 0.7. The divergence is due to the operating system's turning off requests so that the response time and the queue do not increase to infinity.

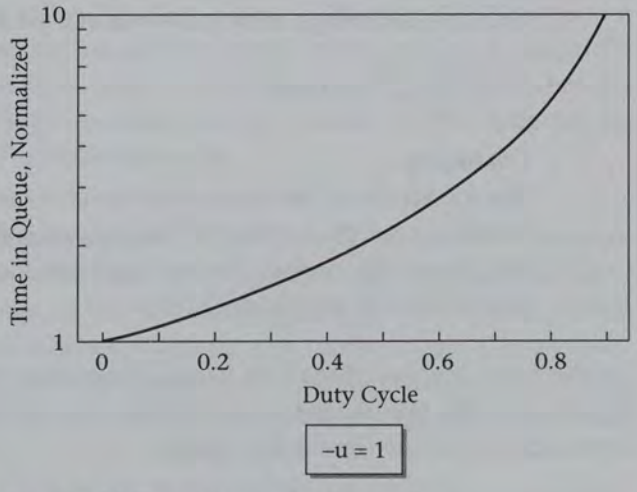


Figure 7.26 Time in queue vs. duty cycle

Petitioner Exhibit 1053, Page 279 of 334

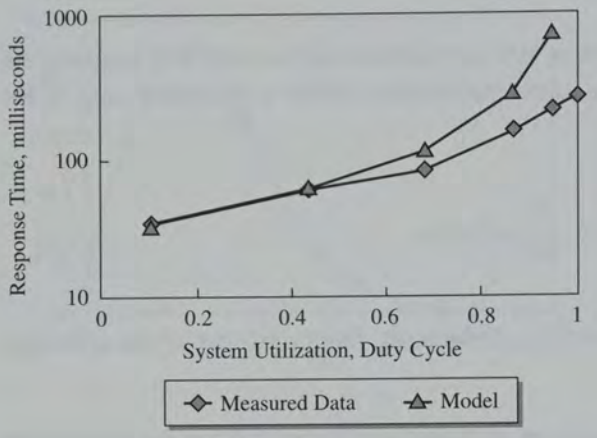


Figure 7.27 Disk system response time

EXAMPLE

An airport departure runway mean departure rate is 0.5 departures per minute or one every two minutes, not counting queue delays. At the present time, the duty cycle of the runway is 50%. There is a projection that the traffic of the airport will increase by 50% in the next year. By what percent will the takeoff delays increase?

Solution

At the present time the mean t_q, including the takeoff time itself, is

$$\frac{1}{\mu(1-\rho)} = \frac{1}{0.5(1-\rho)} = \frac{2}{1-0.5} = 4 \text{ min.}$$

With the increase in traffic, the departure rate does not change and the mean time in the takeoff queue is

$$\frac{2}{1-(0.5 \times 1.5)} = 8 \text{ min.}$$

Thus the percentage of increase in queue time is

$$\frac{100(8-4)}{4} = 100\%.$$

Comments

The model shows the exponential behavior of an open queue. From the initial conditions, a 50% increase in departures results in a 100% increase in the mean total time in the queue. The wait to takeoff increased from a mean of 2 min to a mean of 6 min, a 200% increase.

The mean number of requests in an open system (queue + server) is determined by Little's law. This law states that the number of customers in a system is equal to the arrival rate times the time in the queue:

mean number of requests in system $= \lambda t_q$.

Petitioner Exhibit 1053, Page 280 of 334

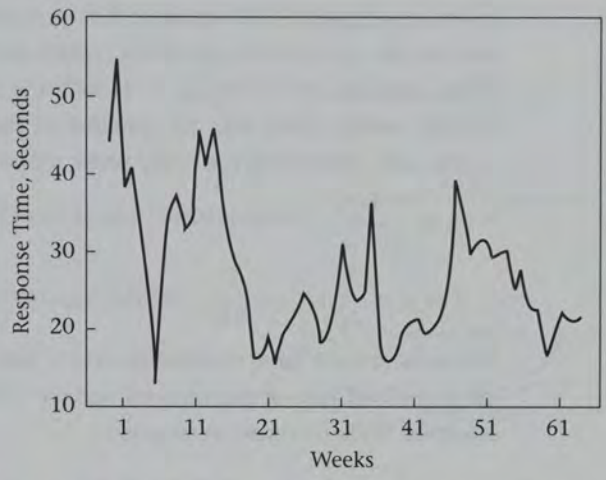


Figure 7.28 WWW response time (data courtesy Keynote Inc.)

Because there are ρ requests being served, the number of requests in the queue is the number of requests in the system minus ρ, that is: $\lambda t_q - \rho$. Thus, as t_q approaches infinity, the size of the queue approaches infinity as well. In real systems, the queue size must be finite, and when the fixed-size queue starts to fill up, requests are denied or turned off. This is what the operating system did with the disk server shown in Figure 7.27.

In the example on the first example of Subsection 7.6.1 what happens if the number of requests to the Internet server jumps to 200 per minute because of a "hot" home page?[72] The depth of the input queue and the queue time will go to ∞, an impossibility. The situation is handled by pushing the queue back to the user with a busy signal, or, in the case of an airport, not letting the planes leave the gates. The queue has not gone away, it has just been pushed back to the user.

EXAMPLES

1. Your boss shows you the following chart, Figure 7.28, of the WWW response time for a 64-week period. You are asked to explain what could have caused the substantial reduction in response time from approximately 55 s to approximately 20 s over this period. There have been periods of short response time but the level achieved at the end seems to be stable.

Solution

To obtain a reasonable answer to this problem, we must make some assumptions. Assume that the response time is for users who have been connected to their provider and does not represent a lack of modems. Also, it is reasonable to assume that the request load increased by 50% during the time in question. We assume that the capacity of the servers was increased substantially; thus the server queue delays were reduced. To find a solution, we assign the total delay

[72] During the 1998 Olympic Games at Nagano, the Internet site had 634,716,480 hits over 16 days, an average of 459 hits per second. The peak hit rate was 1840 per second.

Petitioner Exhibit 1053, Page 281 of 334

to the servers with their queues; that is, assume no communications delay. Also assume an open-system queuing model and that the mean server service time is one request per second ($\mu = 1$) with the overall response time (t_q) of 55 s for the fifth week. What was the speedup of the servers?

We first determine the early week mean request rate:

$$t_q = \frac{1}{\mu - \lambda},$$

$$\lambda = \mu - \frac{1}{t_q} = 1 - \frac{1}{55} = 0.982 \text{ requests per second.}$$

The new servers have improved service rates (mean response time of 20 s) with the increased rate of requests for service. What is the new service rate with the assumed 50% increase in requests?

$$\mu = \frac{1}{t_q} + \lambda = \frac{1}{20} + 0.982 \times 1.5 = 1.52 \text{ services per second.}$$

The speedup of the servers is

$$S = \frac{\mu(\text{new})}{\mu(\text{old})} = \frac{1.52}{1} = 1.52.$$

Comment

It is clear that in the fifth week the servers were greatly overloaded as λ was approaching μ with a duty cycle of $0.982/1 = 0.982$. A 50% improvement in service rate yielded a speedup of $55/20 = 2.75$ in response time and a slight reduction in the server duty cycle to $(\lambda/\mu) = [(0.982 \times 1.5)/1.52] = 0.969$.

The system is still overloaded with this duty cycle.

2. How many requests are waiting in the queue of the Internet server in the fifth and the 64th weeks?

Solution

From Little's law, the mean number of requests in the queue is the total number in the system minus the probability that one request is being served. For the fifth week the mean number of requests in the queue is

$$(\lambda t_q) - \rho = (0.982 \times 55) - 0.982 = 54 - 0.982 \approx 53.$$

From Little's law, the mean number of requests in the queue in the 64th week is

$$(0.982 \times 1.5 \times 20) - 0.969 = 29.46 - 0.98 \approx 29.$$

Comment

The mean number of requests in the queue decreased from 53 to 29, a 45% reduction.

3. Restate Little's law in terms of duty cycle ρ.

Solution

Number of requests in the system $= \lambda t_q$

$$= \lambda\left[\frac{1}{\mu(1-\rho)}\right] = \frac{\rho}{1-\rho}.$$

Petitioner Exhibit 1053, Page 282 of 334

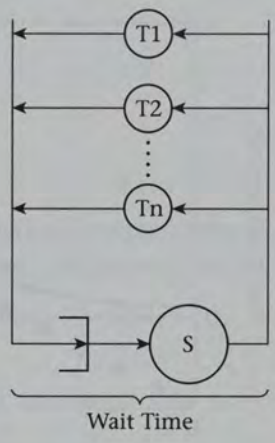


Figure 7.29 Closed-system queuing model

Comment

This statement of Little's law can be checked against the previous solution. The fifth-month duty cycle of the Internet server example is 0.982. Thus the number of requests in the system is $0.982/(1 - 0.982) = 0.982/0.018 \approx 54$, the same result obtained above.

7.6.2 CLOSED-SYSTEM QUEUING MODEL

The second model of interest is the closed-system model. This model provides the mean wait time for a closed system, such as a server with a number of terminals, as shown in Figure 7.29. For this model, there is one server, one input queue, and a number of terminals.

System users compose a request to the server and place the request on the queue. Some time later, the server returns the result of the request, and the user composes another request. Unlike the open system, requests and responses circulate through the system.

We now define some terms: t_s is the server service time, ρ is the duty cycle of the server, n is the number of terminals, t_t is the think time (the time the user takes from the response to the next request), and w is the wait time for the response. The wait time model is

$$w = \frac{nt_s}{\rho} - t_t.$$

The server duty cycle increases from 0 with no active terminals to 1 as the number of requests saturates the server, leaving no idle server time. When the server is saturated, $\rho = 1$, the wait time is

$$w = nt_s - t_t.$$

Note that this is a linear relationship without the exponential increase in service time exhibited with open systems. In addition, without a load on the server, $\rho = 0$ and w becomes very large – an absurd result from the model. There is an iterative solution

Petitioner Exhibit 1053, Page 283 of 334

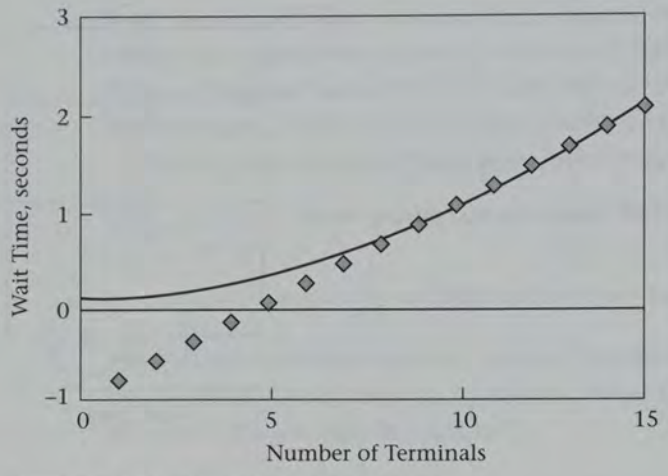


Figure 7.30 Wait time for a closed system

to this model (Denning and Buzen 1978) that removes this absurdity; however, an approximate solution is shown illustrated by assuming some values for the parameters.

Let $t_s = 0.1$ s, $t_t = 1$ s, and $\rho = 1$ (the saturated case). The plot of wait time is shown in Figure 7.30 with the wait time being negative with a small number of terminals, another absurd situation that is eliminated with the iterative solution.

What is the correct wait time plot? For one terminal, there is no other requester and the wait time is merely the service time of the server. This point is plotted at 0.1 s for the example. As the number of terminals increase, ρ increases to 1 and the system is saturated at approximately 10 terminals. As terminals are added, the wait time increases with a slope of t_s. Because many closed systems operate saturated, the saturated region is the region of general interest.

EXAMPLE

A server with 100 terminals is saturated; the measured mean wait time is 1.7 s and the measured mean think time is 2.0 s. The users are complaining about the long wait, and the facility manager wants to reduce the mean wait time to 0.5 s. What can be done?

Solution

There are two solutions to this design problem: the number of terminals can be reduced or a higher-performance server can be provided. To find the number of terminals that will give the desired mean wait time, we must first find t_s of the base system:

$$W = nt_s - t_t,$$
$$1.7 = 100t_s - 2,$$
$$t_s = \frac{1.7 + 2}{100} = 0.037 \text{ s}.$$

To reduce the wait time to 0.5 s, we find the number of terminals that will give

Petitioner Exhibit 1053, Page 284 of 334

that performance:

$$W = nt_s - t_t,$$
$$0.5 = 0.037n - 2,$$
$$n = \frac{0.5 + 2}{0.037} = 67.5 \approx 67 \text{ terminals.}$$

The alternative solution to the long wait problem is to increase the performance of the server. The needed server speedup is found by first determining the required t_s:

$$W = nt_s - t_t,$$
$$0.5 = 100t_s - 2,$$
$$t_s = \frac{0.5 + 2}{100} = 0.025 \text{ s.}$$

The required speedup is

$$\text{Speed up} = \frac{t_s(\text{old})}{t_s(\text{new})} = \frac{0.037}{0.025} = 1.48.$$

Comment

The facilities manager will need to drop 1/3 of the existing terminals to give the desired response time, not a very likely solution. The most likely solution is to upgrade the server by approximately 50%.

A word of caution is needed regarding think time. Think time is presented in the model as a fixed value. There is ample experimental evidence – and your own experience will tell you – that as the wait time decreases, think time decreases as well. We make more requests to a fast system than to a slow one. Thus the facilities manager, in the example above, should compute the required speedup by using $t_t = 1$ rather than $t_t = 2$.

REFERENCES

Blaauw, G. and Brooks, F. P. Jr. (1997). *Computer Architecture, Concepts and Evolution*, Addison-Wesley, Reading, MA.

Chaney, T. J. and Molnar, C. E. (1973). "Anomalous Behavior of Synchronizer and Arbiter Circuits," *IEEE Transactions on Computers*, **C-22**:4.

Chen, P. (1989) "An Evaluation of Redundant Arrays of Inexpensive Disks Using an Amdahl 5890," M. S. Thesis, Technical Report UCB/CSD 89/506, p. 341, Computer Science Division, University of California, Berkelay.

Del Corso, D., Kirmann, H., and Nicoud, J. D. (1986). *Microcomputer Buses and Links*, Academic, New York.

Denning, P. J. and Buzen, J. P. (1978). "The Operational Analysis of Queuing Network Models," *ACM Computing Surveys*, **10**:3.

Di Giacomo, J., ed. (1990). *Digital Bus Handbook*, McGraw-Hill, New York.

Kleeman, L. and Cantoni A. (1987). "Metastable Behavior in Digital Systems," *IEEE Design and Test*, **4**:6.

Randell, B., ed. (1973). *The Origins of Digital Computer: Selected Papers*, Springer-Verlag, New York, 1973.

Petitioner Exhibit 1053, Page 285 of 334

Thurber, K. J., Jensen, E. D., Jack, L. A., Kenney, L. L., Patton, P. G., and Anderson, L. C. (1972). "A Systematic Approach to the Design of Digital Bussing Structures," *Fall Joint Computer Conference*, AFIPS Press, Montvale, NJ, pp. 719–736.

EXERCISES

7.1 Using the example above on WWW response time, will the result be significantly different if the estimate of 1 s for the service time is changed to 0.1 s? Work out the solution.

7.2 Write a program using the von Neumann ISA, Table 1.2, for maintaining a counter in a memory location to control a programmed I/O transfer.

7.3 Derive a model for the percentage of increase in the time in the queue that is a function of an increase in ρ from ρ_1 to ρ_2. Assume that μ does not change. Verify your solution with the example found with the airport departure runway example.

7.4 A computer's bus is 0.5 m long and the card holding the clock generator is placed in the center of the bus. What deskewing delay, in nanoseconds, is required on the clock feeding the logic on this center board? You must make an assumption as to the propagation delay down the bus for terminated lines.

7.5 For the example on terminals, in Subsection 7.6.2, recompute the server speedup needed for $t_t = 1$ rather than $t_t = 2$.

7.6 Consult the documentation for the Intel 8080 and list the complete set of codes for the three pins noted in Section 7.1.

7.7 Refer to Figure 7.13 in which deskewing delays are inserted in the clock lines. What is the magnitude of the delay for each card slot under the assumption that there can be five cards on the bus?

7.8 Why is distribution of a skew-free clock over a serial transmission path impractical? This issue is discussed in Section 7.4.

7.9 You are shown the plot of Figure 7.25. The distribution seems to be Poisson. From these data compute the value of λ.

7.10 Find the server speedup required in the last example on example in Subsection 7.6.2 if think time decreases to 1 s when the mean response time is decreased to 0.5 s.

7.11 The programmed I/O example assumed a processor with a 0.1-MIPS instruction processing rate. Rework this example with MIPS = 0.1, 1, 10, and 100. Plot your results of efficiency versus MIPS.

7.12 The example in Section 7.2 shows the system providing 12.35 transactions per second. Assume a requirement of 500 transactions per second. The processor will be unchanged but additional disks will be added. How many disks are required?

Petitioner Exhibit 1053, Page 286 of 334

EIGHT

EXTENDED INSTRUCTION SET ARCHITECTURES

8.0 INTRODUCTION

User ISAs, discussed in Chapter 3, provide operations, addresses, and data types for many types of scientific and business applications. However, these instructions are insufficient for processors used in modern systems with operating systems, virtual memory, concurrent I/O, and in multiprocessor configurations. Thus ISAs are extended to provide support for these additional tasks. Each of these extensions to the ISA requires special data types, addresses, and operations.

The programs that provide these extended services are usually written by system software providers, not by the user. To prevent user modification or tampering with these programs, the processors operate in one of two modes: user mode or supervisor mode.

The extensions to the ISA are collectively called control instructions in the IBM S360 documentation (IBM 1970), protected mode instructions in the Intel Pentium (Intel Corporation 1993), and privileged instructions in the Pentium Pro documentation (Intel Corporation 1996). In this text we use the term privileged instructions.

Privileged instructions cannot be executed along with the user instructions when the processor is in user mode. However, when in the supervisor mode, both user and privileged instructions can be executed. If a privileged instruction is attempted when the processor is in the user mode, an interrupt is generated in the decode stage of the pipeline. The separation between the privileged instructions and the user instructions is enforced in many computers by means of a mode control switch. The instructions that switch from supervisor mode to user mode are privileged, but a user instruction can invoke privileged mode.

EXAMPLE

Give an example of transfer from user mode to supervisor mode and back to user mode.

Solution

With the IBM S360, S370, and S390, the supervisor call instruction (SVC) is a user instruction. This instruction causes an interrupt to the supervisor, and the supervisor loads a new program status word (PSW) from memory location 96

Petitioner Exhibit 1053, Page 287 of 334

that has a control bit (bit 15) set to supervisor mode.[73] The user PSW is saved in memory location 32. After the supervisor call interrupt has been serviced, the privileged load PSW (LPSW) instruction restores the old PSW and restarts the interrupted program.

Comment
If the PR instruction is executed and the processor is not in the supervisor mode, an illegal op-code interrupt will be generated when the instruction is decoded.

The IBM S390 introduces semiprivileged instructions that can be executed in the user mode when certain authority requirements are met. Likewise, the Intel Pentium Pro has four levels of privilege. Level 0 is the operating system kernel, levels 1 and 2 are the operating system services (device drivers, etc.), and level 3 is the user or application level.

In this chapter, we discuss the privileged instructions, their operations, data types, and addresses. Two classes of extended user instructions are also discussed in Section 8.7: multiprocessor support and MMX enhancements to the Intel Pentium.

8.1 OPERATING SYSTEM KERNEL SUPPORT

A subset of privileged instructions, operating system kernel support, is discussed in this section. There are some operating system instructions that are similar to user instructions except that they use different registers. Processors may have a set of registers that are addressed only by the privileged instructions. Thus there are, in essence, other processors embedded in the user processor. However, there are common resources such as caches, ALUs, and implemented registers used by all instructions.

The processor state consists of the system data types that are collections of bits, fields of bits, and words allocated to registers that are used as semifixed indicators of some state of the process and processor. These registers hold indicator bits that define at any time the current operating mode and conditions of the processor. The processor state registers usually include the indicator bits, discussed in Chapter 3 in the context of branching. In addition to the state register data types, the memory addresses associated with these data types are described along with the operations that can be performed on the data types. The processor state registers for the IBM S370 and the Intel Pentium Pro are

IBM S370 (IBM 1976):
- PSW
- Control registers

Intel Pentium Pro (Intel 1996):
- Flag register
- Control registers
- Instruction pointer register

[73] IBM calls this bit *P* for problem state bit.

Petitioner Exhibit 1053, Page 288 of 334

The PSW, flag register, and the instruction pointer register hold bits that may be changed dynamically during program execution. Control registers hold bits that are not normally modified during program execution. These bits establish the mode of operation and are swapped to/from memory because of a context switch or, with some processors, a subroutine call.

The data types described in this chapter have many bits and fields allocated to computer words. Students need not memorize these allocations, as detailed knowledge can always be obtained from the manufacturers' literature. However, students should understand why the data types exist and have a general understanding of the function and the purpose of the data types.

The operating system kernel data types exist to control the system, not produce results. Thus they never leave the system as outputs. The data types are transferred to dedicated registers in the processor, where they perform their intended function. These data types may be modified by privileged instructions during program execution or by side effects to user instructions.

IBM S360, S370, and S390

The IBM S360 family of computers was designed from the beginning to be controlled by an operating system. Thus a number of hardware and software concepts were introduced that facilitate the design and use of an operating system. We look at the data types, addresses, and operations that support the operating system kernel.

Data Types. Figure 8.1 shows the PSWs for the IBM S360 basic control (BC) mode and the S370 extended control (EC) Mode. The S370 extended the S360 with virtual memory requiring additions to the S360 PSW. Note that the program counter, called the instruction address, is found in the PSW as is the 2 bit conditing code (CC). Bits 0–7 set the control of the interrupts.

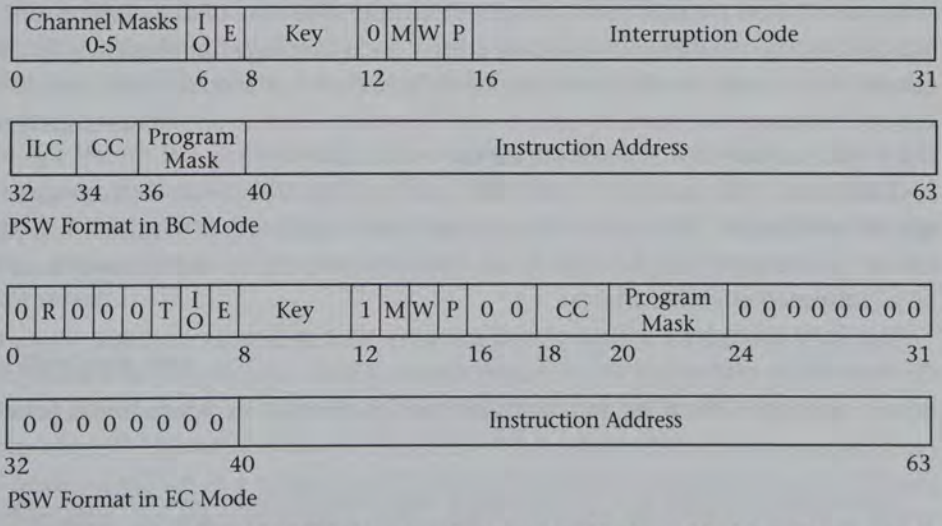


Figure 8.1 IBM S360, S370 PSWs (courtesy International Business Machines Corporation)

Petitioner Exhibit 1053, Page 289 of 334

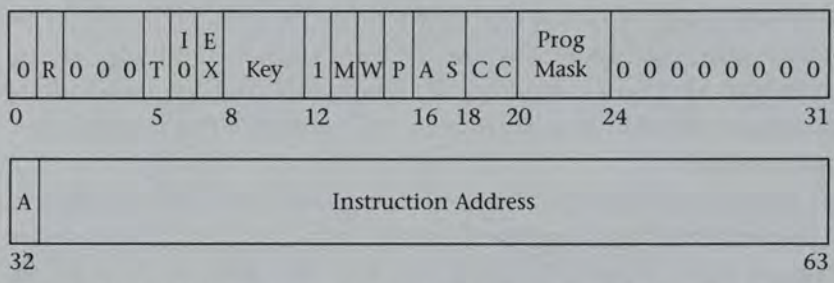


Figure 8.2 IBM S390 PSW (courtesy International Business Machines Corporation)

A 4-bit key provides protection for 2048-byte blocks in memory.[74] Each block in memory has an associated 7-bit key, called key in storage. For CPU writes, the 4-bit key in the PSW must match the corresponding 4-bit key in storage. CPU reads are permitted if the key matches or if bit 4 in the key in storage is zero. This key mechanism provides protection from unauthorized reads and writes. The key in the PSW is modified by a privileged instruction.

The 4-bit program mask augments the condition code for fixed-point overflow, decimal overflow, floating-point underflow, and floating-point significance when the result fraction of a floating-point operation is zero. An interrupt is generated when any of the mask bits are set to 1. The operating system kernel then determines the source of the interrupt by examination of the condition code, key, and/or the interrupt code.

The S390 PSW, EC mode, reallocates all the bits except the instruction address, key, and a few other bits, as shown in Figure 8.2. The major difference from the S370 is that the instruction address has been increased to 31 bits from 24. Reference should be made to an operations manual for a full description of the bits.

In addition to the PSW data type, the IBM S370 has a 16-word 32-bit control register (CR) file.[75] One or more bits in these registers is assigned to functions needing control or specification. Some of the bits are assigned to virtual-memory address translation and others to multiprocessing (for example, the time of day clock mask). Other bits control the recording of program events, such as successful branches. Finally, bits control how machine checks are handled: logout or not logout. Table 8.1 shows the general functions allocated to each of the CR words. Refer to any detailed architectural description of the IBM S370 to obtain the exact allocations.

Addresses. The contents of the PSW and the CRs are moved to/from memory by special instructions. Thus a source and a destination address are required. The PSW and CRs are addressed by implication in the instruction op-code, and memory is addressed by a computed effective address.

The S370 provides a region of real memory that is directly accessed – i.e., the addresses are not mapped by the virtual-memory system. This region of memory, 0–1FC, is used to store PSWs, CRs, and machine-check information. Space is also provided to

[74] A block may or may not be a virtual-memory page as the S370 can have 4096-byte pages.
[75] The S360 did not have CRs; they were added with the S370 and their use expanded with the S390.

Petitioner Exhibit 1053, Page 290 of 334

TABLE 8.1 IBM S370 CR ALLOCATIONS

Word	Primary Use
0	Multiprocessing, dynamic address translation, timer
1	Dynamic address translation
2	Channel masks
3	Unassigned
4	Unassigned
5	Unassigned
6	Unassigned
7	Unassigned
8	Monitor
9	Program-event recording
10	Program-event recording
11	Program-event recording
12	Unassigned
13	Unassigned
14	Machine-check handling
15	Machine-check handling

store floating-point and general register values. The reason this region is direct mapped is to ensure that its contents are never paged out to the disk; the contents are always directly accessible by the processor.

Operations. Before the operations are described, it is instructive to examine the six instruction formats of the IBM S370 shown in Figure 8.3. The instructions can be 2, 4, or 6 bytes long. A wide variety of addressing options are provided. The formats associated with each of the instructions of interest are indicated in the text by their two-character format codes, such as RR for register to register.

The IBM S360 has 11 privileged instructions: 5 for the kernel (plus 1 nonprivileged instruction), 4 for I/O, and 2 for multiprocessing. The IBM S360 set of instructions is a smaller set than the 26 of the IBM S370 and the 55 privileged and semiprivileged instructions of the IBM S390. For instructional purposes, the simple S360 set is described.

Table 8.2 shows the instructions that are used to modify the status of the CPU by loading and storing the bits in the PSW. Note that the storage key can be loaded or stored but the system mask and the program mask can only be loaded into the PSW. The diagnose instruction invokes hardware diagnostics that are unique to the processor model.

Intel Pentium Pro

The Intel Pentium Pro evolved from the Intel 8008, which was not designed to be under the control of an operating system. Thus some of the features have the appearance of being added on to a simpler architecture. The operating system has four levels of privilege with processor state data types, addresses, and operations for the support of the operating system kernel.

Petitioner Exhibit 1053, Page 291 of 334

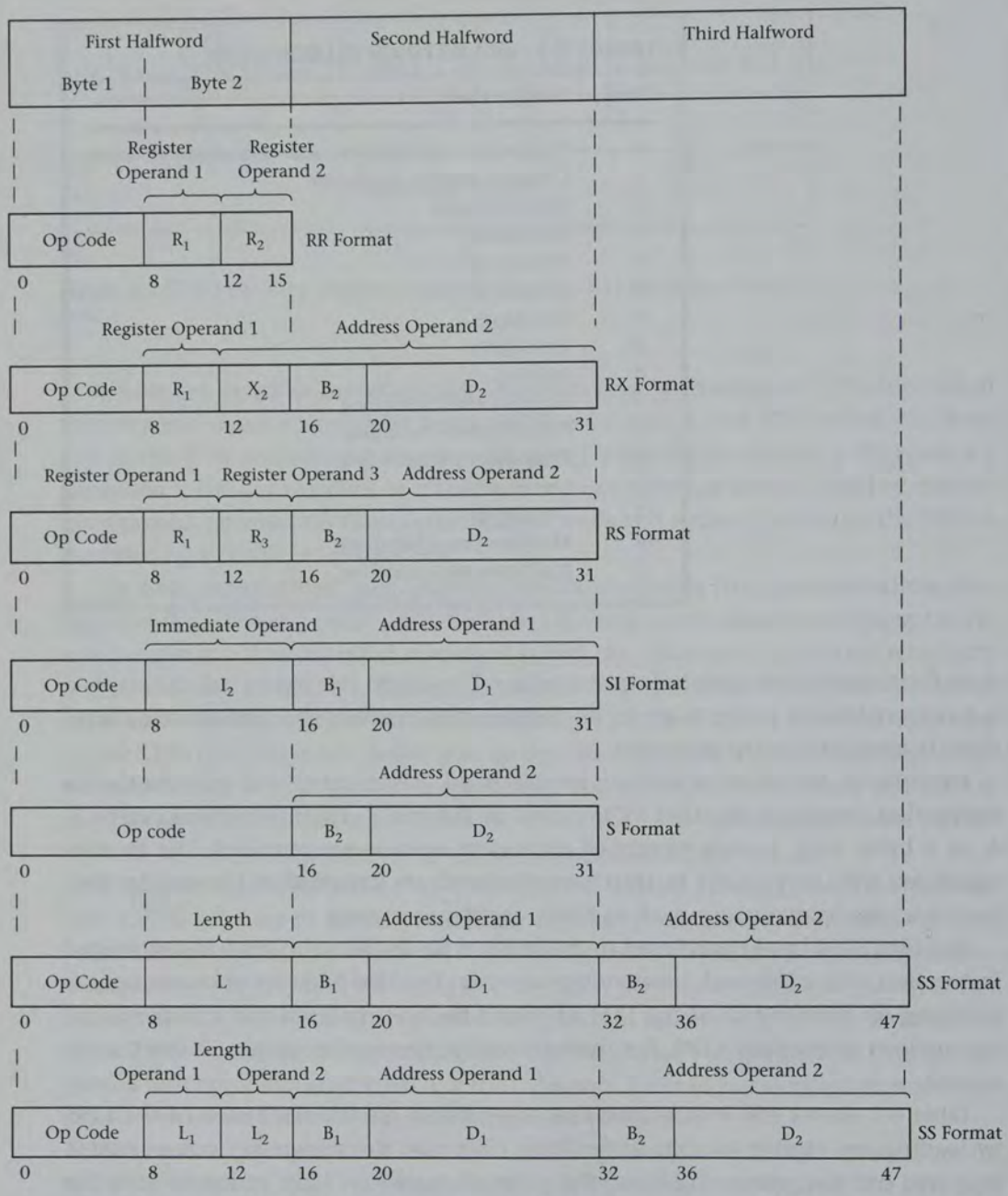


Figure 8.3 IBM S370 instruction formats (courtesy International Business Machines Corporation)

Data Types. The processor state data types of the Intel Pentium Pro consist of bits allocated to the instruction pointer (EIP) register, the flag register (EFLAG), also known as indicator bits, and four CRs.[76] These registers are in hardware and establish the

[76] The addition of the letter "E" to IP and FLAG register names occurred with the Intel 386 when these registers were extended to 32 bits from 16 bits of the Intel 286.

Petitioner Exhibit 1053, Page 292 of 334

TABLE 8.2 IBM S360 STATUS MODIFICATION INSTRUCTIONS

Name	Mnemonic (Format)	Function
Insert storage key*	ISK (RR)	The storage key associated with the block in memory addressed by R_2 is placed in bits 24–28 of the register addressed by R_1.
Set storage key*	SSK (RR)	Bits 24–28 of the register addressed by R_1 are placed in the storage key associated with the block of memory addressed by R_2.
Load PSW *	LPSW (SI)	The current PSW is replaced with the double word $m(D_2 + (B_2))$. The current PSW is not retained.
Set system mask*	SSM (SI)	The addressed byte replaces the system mask of the current PSW.
Set program mask	SPM (RR)	Bits 2–7 of the register addressed by R_1 replace the condition code and program mask of the current PSW.
Diagnose*	None	The CPU performs builtin hardware diagnostic tests specific to the processor model.

* Privileged.

sequencing of instructions, the operating modes of the processor and assist in the translation of the virtual-memory address.

The EIP is a 32-bit data type that points to the first byte of an instruction. The EIP is incremented after each instruction is decoded; the size of the increment is equal to the number of bytes in the current instruction.

Figure 8.4 shows the bit allocation to the flag register. Note that bits 0–7 have been discussed in Chapter 3, Figure 3.12, in the context of indicators for branching. The details regarding the use of the flag bits can be found in an Intel Pentium Pro user's manual.

The four Pentium Pro CRs are shown in Figure 8.5. We note a few interesting aspects of these registers. The page directory base in CR3 was discussed in Chapter 4, Figure 4.6, in the context of extending the linear address to 52 bits. Also note that CR2 holds the page fault linear address that is used to reference memory for the page fault handler.

Addresses. The instruction pointer, flag register, and CRs are implemented in hardware and are loaded from and stored to memory by means of the general registers, shown in Figure 3.8. Two addresses are required, the source and the sink address. For both the flag register and the CRs, both the addresses are implicit in the instruction op-code. The instruction pointer is addressed by implication by means of the op-code when it is stored during a context switch.

Operations. The only ISA operation for the flag register and the CRs are special move instructions, shown in Table 8.3. Status flag bits are moved to/from memory by means of register AH, and CR data types are moved to/from memory by means of a general-purpose register, shown in Figure 3.8. The flag register can also be pushed

Petitioner Exhibit 1053, Page 293 of 334

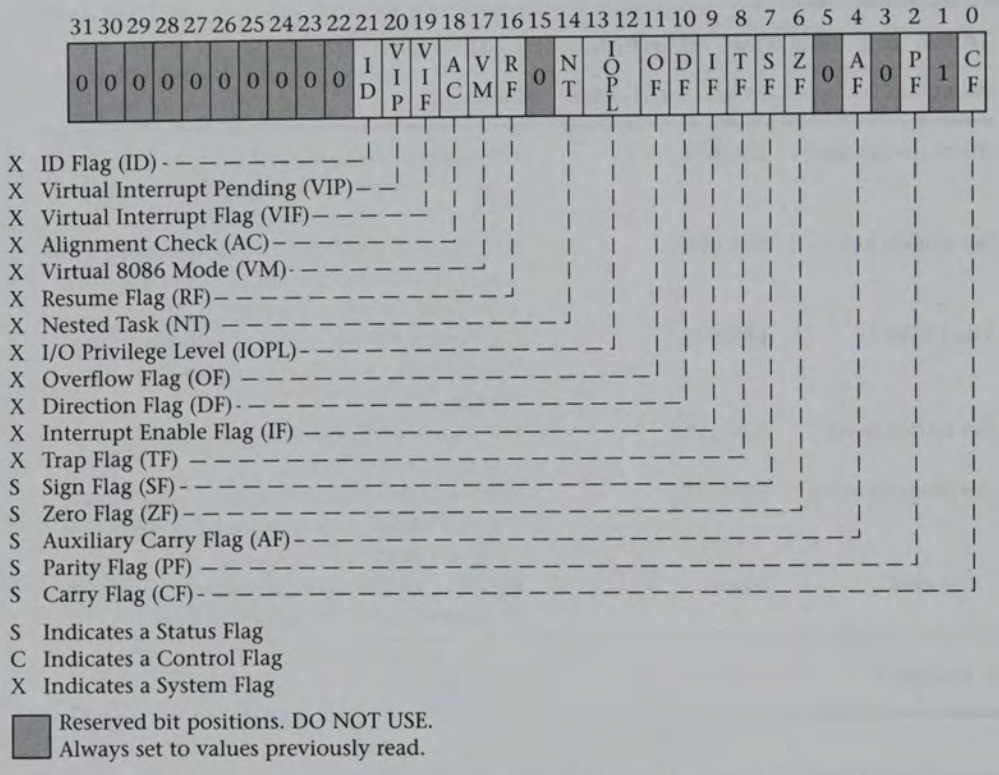


Figure 8.4 Intel Pentium Pro flag register (courtesy of Intel Corporation)

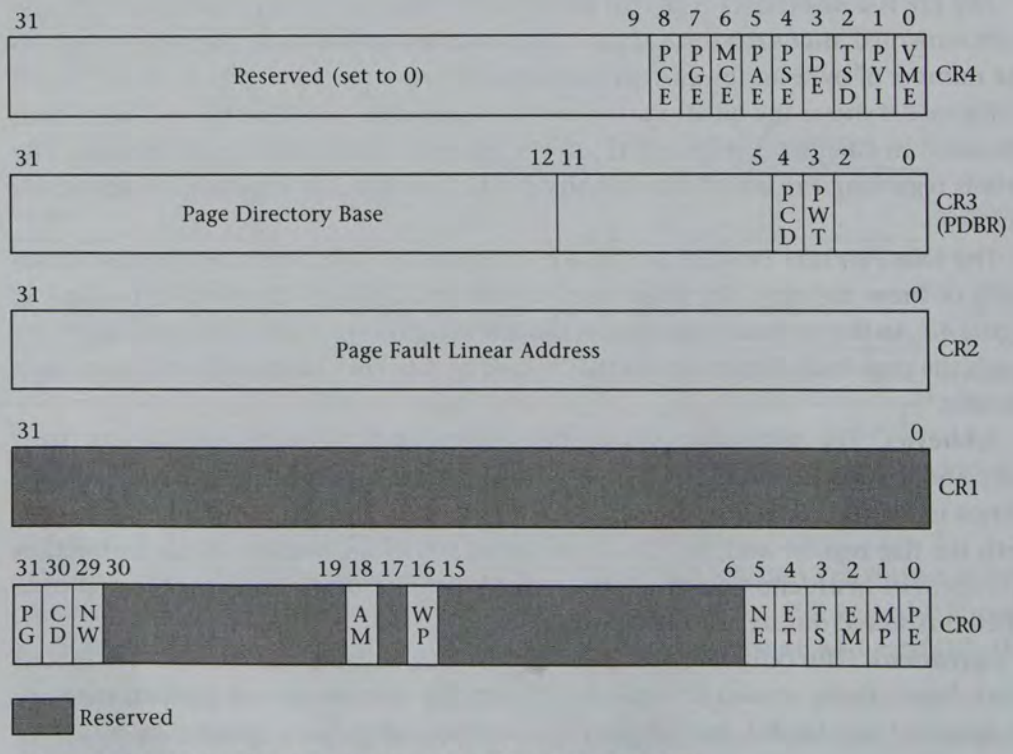


Figure 8.5 Intel Pentium Pro CRs (courtesy of Intel Corporation)

TABLE 8.3 FLAG REGISTER AND CR INSTRUCTIONS

Name	Mnemonic	Function
Load status flag into AH register	LAHF	The low bite of the EFAGS register is moved to the AH register.
Store AH into flags	SAHF	The SF, ZF, AF, PF, and CF flags are moved to the flag register. Bits 1, 3, and 5 are unchanged.
Push EFLAGS onto the procedure stack*	PUSHF	Pushes the contents of the flag register onto the procedure stack and decrements the stack pointer.
Pop the procedure stack into the flag register*	POPF	Pops the top of the procedure stack onto the flag register and increments the stack pointer.
Move to/from CRs*	MOV CRn	Moves 32-bit words between general-purpose register and one of the CRs. There are eight of these instructions.

* Privileged.

and popped to/from the procedure stack. Operations on the instruction pointer are described in the section on interrupt handling.

Note that some of the bits in the flag register are set/reset as side effects of the user instruction set. With the move instructions, the state of the flag register can be saved by move instructions executed for a context switch.

8.2 VIRTUAL-MEMORY SUPPORT

The primary task of the virtual-memory support ISA is the allocation, protection, and management of the segment tables and page tables that are used to translate virtual addresses to real-memory addresses. Chapter 4 described the overall operation of paged and segmented virtual-memory systems.

IBM S370

The primary difference between the IBM S370 and the IBM S360 was the addition of virtual memory. Because of tight hardware and software development schedules, virtual memory was not included in the S360.[77] The IBM literature refers to the virtual

[77] The S360 Model 67, a precursor to the S370, provided virtual memory.

Petitioner Exhibit 1053, Page 295 of 334

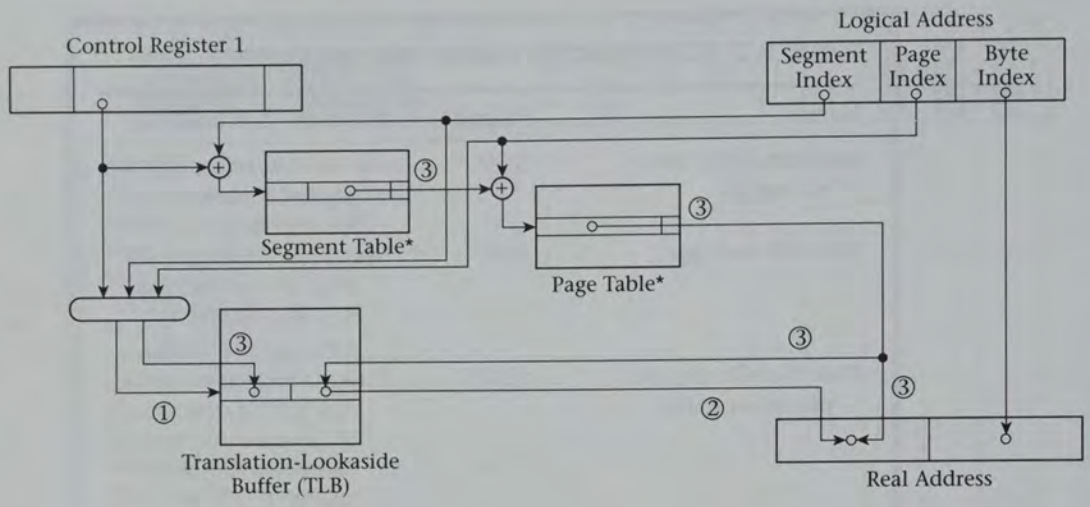


Figure 8.6 S370 Virtual-address translation (courtesy International Business Machines Corporation)

memory as dynamic address translation. The PSW in the EC mode has bit 5 (T), shown in Figure 8.1, that invokes dynamic address translation when set. If not set, all effective addresses are direct real-memory addresses. Figure 8.6, similar to Figure 4.35, shows the IBM S370 paged segmented address translation process.

Data Types. Four data types are required for address translation: the segment table address in CR 1, the page table address in the segment table, the page address in the page table, and the TLB entries, all shown in Figure 8.7. The CRs and the TLB are implemented in hardware whereas the segment table and the page table reside in real memory. Recall from Chapter 4 that the fields of the logical address vary in length depending on the size of the segment and the size of the page.

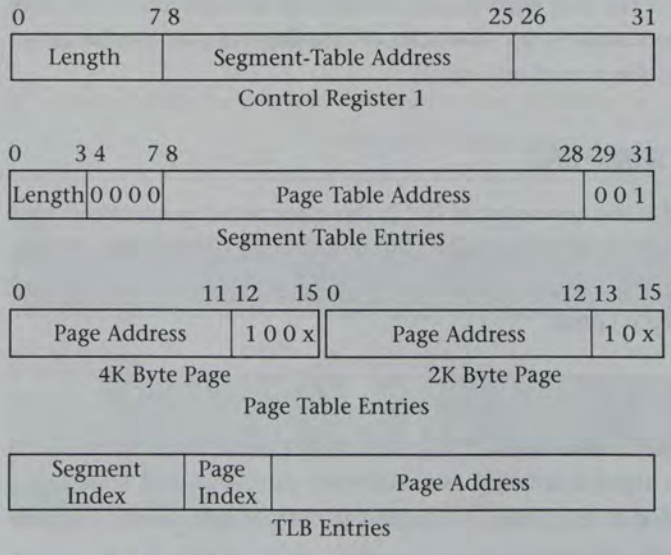


Figure 8.7 IBM S370 virtual-memory data types

Petitioner Exhibit 1053, Page 296 of 334

In CR 1, there are two allocated fields, an 18-bit segment table address and an 8-bit segment length field; a segment can be as large as 2^8 pages. Six bits are unallocated. The index into the page table is formed by adding the segment table address, in CR 1, to the segment index, from the logical address.

The segment table entry contains the 21-bit page table address, which is added to the page index field from the logical address, producing an index into the page table. The 4-bit field holds the length of the page table in units of 1/16 of the maximum size of the table. The value is compared with the page index field of the logical address to ensure that a reference is not made to an out-of-bounds page table entry.

The IBM S370 supports two page sizes, 2K and 4K bytes, which are selected in CR 0 by bits 8 and 9. The page table data types are also shown in Figure 8.7. Each page table entry is 16 bits. Bit 12 or 13, depending on page size, is the present bit for the page. If the present bit is a 0, the translation proceeds. If the present bit is a 1, a page translation interrupt is generated. The page address is concatenated with the byte index field of the logical address to form the real-memory address.

The TLB data type entries consist of the tag – portions of the segment index and the page index – to establish the presence of the entry and the real-memory page address. The size and the organization of the TLB depend on the particular IBM S370 model.

Addresses. The virtual-memory support data types are allocated to spaces with addresses, as shown in Table 8.4. Note that untranslated real-memory addresses are used for the segment table and page table entries. The reason for this is to position these tables in memory so that page faults will not occur when they are accessed serving a page fault.

Operations. There are two privileged instructions for explicitly loading and storing CRs, as shown in Table 8.5. A nonprivileged move instruction (from the user instruction set) moves the segment table and the page table entries into their proper place in memory. The PTLB instruction purges or clears the TLB, and the SPX instruction achieves the same end as a side effect.

Intel Pentium Pro

The Pentium Pro uses a paged segmented virtual-memory system, described in Section 4.6. The virtual-address generation and address translation system for 4K-byte pages is shown in Figure 8.8. This figure is similar to Figure 4.36. Another virtual-address translation configuration is provided for 4M-byte pages; the selection between the two is made by bit 4 (PSE) in CR 4, shown in Figure 8.5.

TABLE 8.4 IBM S370 VIRTUAL-MEMORY SUPPORT DATA TYPES

Data Type	Allocation	Address
CR 1	Hardware register	Implied in op-code
Segment table entry	Real memory	Untranslated real-memory address
Page table entry	Real memory	Untranslated real-memory address
TLB	Hardware registers	Implied in instructions that handle page faults

Petitioner Exhibit 1053, Page 297 of 334

TABLE 8.5 IBM S370 VIRTUAL-MEMORY SUPPORT INSTRUCTIONS

Name	Mnemonic (Format)	Function
Load control*	LCTI (RS)	The set of control registers beginning with (R_1) through (R_3) are loaded from $m(D_2 + (B_2))$.
Store control*	STCTL (RS)	The set of control registers beginning with (R_1) through (R_3) are stored in $m(D_2 + (B_2))$.
Move	MVC (SS)	The number of bytes L are moved from $m(D_1 + (B_1))$ to $m(D_2 + (B_2))$.
Purge TLB*	PTLB (S)	All information in the TLB is marked invalid.
Set prefix *	SPX (S)	SPX is used with multiprocessor configurations. The TLB is marked invalid as a side effect.

* Privileged.

Data Types. There are six data types for the paged segmented Pentium Pro virtual memory. These are the segment selectors in the segment registers, CR 3, the segment descriptor table entries, the page directory table entries, page table entries, and the TLB entries.

Figure 8.9 shows the segment selector data type. This data type is held in hardware registers, as described in Section 4.6, and contains the 14-bit index into the segment

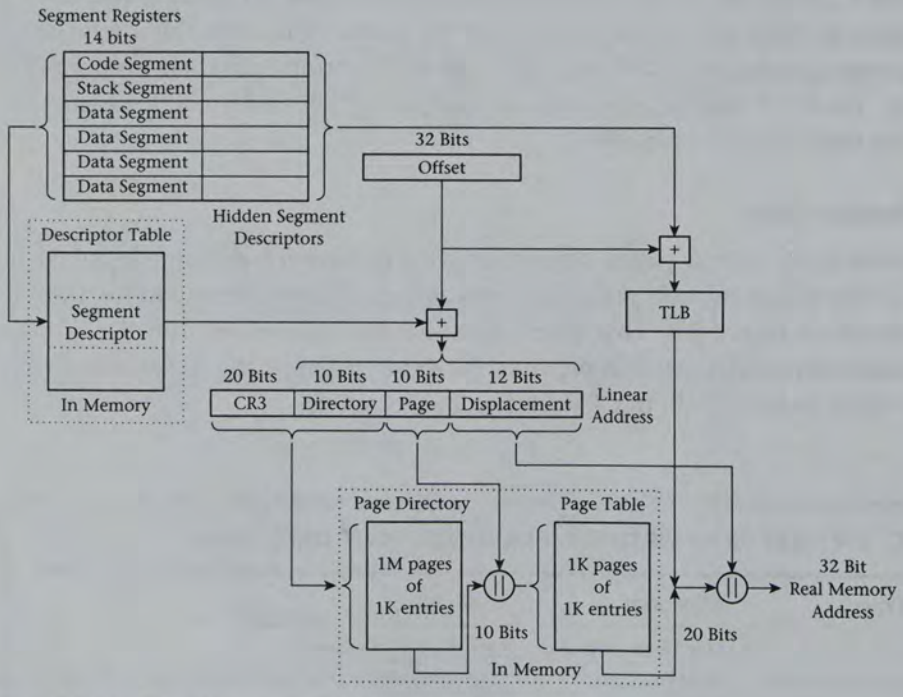


Figure 8.8 Pentium Pro segmented virtual-address system

Petitioner Exhibit 1053, Page 298 of 334

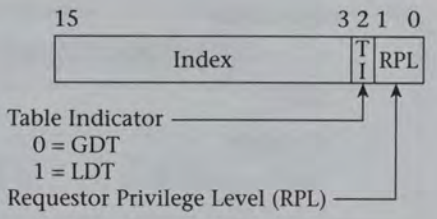


Figure 8.9 Segment selector (courtesy of Intel Corporation)

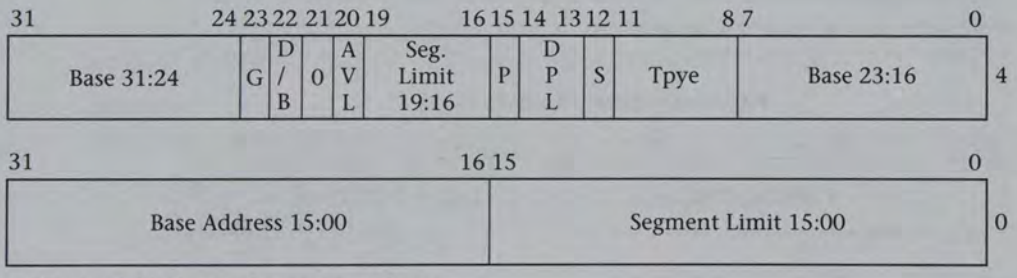
descriptor table. There are two segment descriptor tables, global and local, selected by bit 2. The two RPL bits select one of the four access privilege levels, discussed in Section 8.0.

Allocations to CRs are shown in Figure 8.5. The 20 MSBs of CR 3 extend the linear address to 52 bits. The PCD and PWT bits control caching of the page directory and page table entries in the processor's data cache.

Figure 8.10 shows the segment descriptor data type. The data type is allocated to 8 bytes, indicated by 4 and 0. The two major components of the segment descriptor are the base address of the page directory and the size limit of the segment. The 32-bit base address is formed by bits 16–31 of word 0, bits 0–7, and bits 24–31 of word 4. Likewise, the segment limit is formed from bits 0–15 of word 0 and bits 16–19 of word 4, giving a 20-bit segment limit.

Note the G-Granularity flag in the segment descriptor. If the G flag is 0, the segment size can vary between 1 byte and 1M bytes in 1-byte steps. If G is 1, the segment size can vary between 4K bytes and 4G bytes in 4K-byte (one-page) steps.

If the value of the 20 MSBs of the processor's 32-bit offset or effective address is greater than the value of the 20-bit segment limit field, an addressing error interrupt will be generated, which must be handled by the operating system.



AVL–Available for use by system software
BASE–Segment base address
D/B–Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL–Descriptor privilege level
G–Granularity
LIMIT–Segment Limit
P–Segment present
S–Descriptor type (0 = system; 1 = application)
TYPE–Segment type

Figure 8.10 Pentium Pro segment descriptor data type (courtesy of Intel Corporation)

EXAMPLE

Show the calculation for the maximum segment size for the two cases of granularity.

Solution

The segment limit is 2^{20} of the granularity increments. Thus,

For $S = 0$ size (1-byte increment) $= 1 \times 2^{20} = 2^{20} = 1$M byte,
For $S = 1$ size (4K-byte increment) $= 2^{12} \times 2^{20} = 2^{32} = 4$G bytes.

Recall from Figure 8.8 that the linear address is formed by adding the 32-bit segment descriptor base address to the 32-bit offset. The 10 MSBs of the linear address are concatenated with 20 bits from CR 3 to form the index into the page directory, as also shown in Figure 4.6.

The page directory and the page table entries data types are shown in Figure 8.11. The page directory is indexed to find the page table base address. The page table base address is concatenated with 10 bits from the linear address to form the index into the page table.

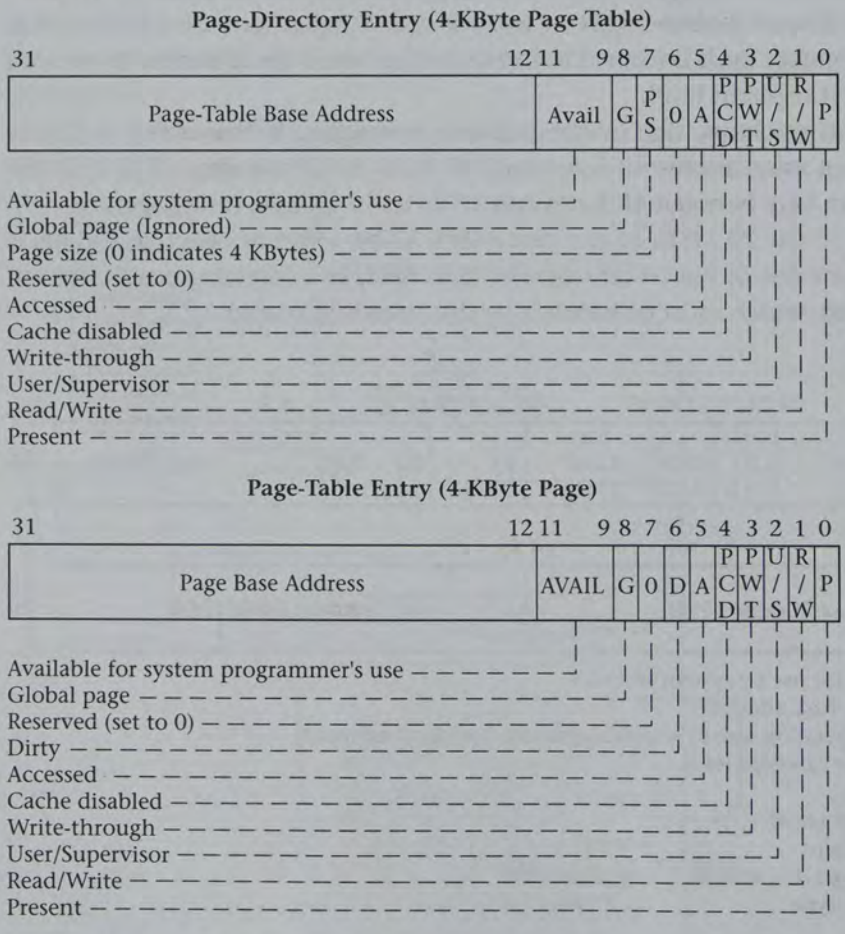


Figure 8.11 Pentium Pro page directory and page table entries (courtesy of Intel Corporation)

TABLE 8.6 PENTIUM PRO VIRTUAL-MEMORY SUPPORT DATA TYPES

Data Type	Allocation	Address
Segment selector	Hardware register	Implied by the memory access type: CS, SS, DS, ES, FS, and GS
CR 3	Hardware register	Automatically concatenated with the linear address. Implied in op-code or as a destination for a move
Segment descriptor	Real memory	Untranslated real memory
Page directory table	Real memory	Untranslated real memory
Page table entry	Real memory	Untranslated real memory
TLB	Hardware registers	Implied in instructions that handle page faults

The page directory entry and the page table entry are quite similar, providing a 20-page table base address and a 20-bit page base address. Note the P-Present bits. The P bit in the page-directory entry signifies that the entry in the page table is or is not present, whereas the P bit in the page table entry signifies whether or not the page itself is present in real memory.

The organization of the Pentium Pro TLB is shown in Figure 4.21. The data type is the pretranslated real-memory page base address; the same as found in the page table entry. A valid bit permits each entry to be invalidated by a privileged instruction.

Addresses. The six virtual-memory support data types are allocated to spaces with addresses, as shown in Table 8.6.

EXAMPLE

With the descriptor table, page directory, and page table allocated to real memory, what is the maximum percent of real-memory space that can be occupied by these tables? The total real-memory address space is 2^{32} bytes.

Solution

The descriptor table is indexed by a 14-bit pointer, and the segment descriptor is 4 bytes. Thus the maximum percent of real-memory space occupied is

$$\frac{100 \times 2^{14} \times 2^2}{2^{32}} = 0.0015\%.$$

The page directory is indexed by a 30-bit pointer, and the page directory entries are 4 bytes. Thus the maximum percent of real-memory space occupied is

$$\frac{100 \times 2^{30} \times 2^2}{2^{32}} = 100\%.$$

The page table is indexed by a 20-bit pointer, and the page table entries are 4 bytes. Thus the maximum percent of real-memory space occupied is

$$\frac{100 \times 2^{20} \times 2^2}{2^{32}} = \frac{100 \times 2^{22}}{2^{32}} = 0.09\%.$$

Petitioner Exhibit 1053, Page 301 of 334

TABLE 8.7 INTEL PENTIUM PRO VIRTUAL-MEMORY SUPPORT INSTRUCTIONS

Name	Mnemonic	Function
Move to/from segment selector register	MOV	Moves 16-bit words between a segment register and memory
Invalidate TLB entry*	INVLPG	Invalidates the TLB entry specified by the source memory operand
Move to/from CR	MOV CR3	Invalidates all TLBs when there is a write to CR3
Return from procedure	RET	CS segment selector register is loaded as a side effect of the RET instruction.

* Privileged.

Operations. The instructions for managing the virtual-memory data types are shown in Table 8.7. Because the segment descriptor, page directory, and page table data types are stored in real memory, only nonprivileged user move instructions are required for loading and storing these data types. However, interrupts will be generated if the move instruction is used with various CR bits set and the processor is in protected mode or real-address mode. As the CR bits are set with a privileged instruction, the net effect is that these moves are indirectly privileged.

Five segment selectors, other than CS, are loaded with a MOV instruction. TLB entries are loaded only as side effects of page fault processing. However, the privileged instruction INVLPG is provided for invalidating TLB entries. The TLB is also invalidated as a side effect of some other instructions. The TLB is never stored in memory, even with a context switch.

8.3 INTERRUPT SUPPORT

The general principles of interrupts have been discussed in Chapter 3. In this section, the specifics of the instruction set extensions for the IBM S360 and the Intel Pentium Pro are described.

IBM S360

The issues of the source of interrupts and how interrupts are handled have been discussed previously. The IBM S360 – hardwired control, microprogrammed control, and pipelined – handles interrupts along the lines discussed. The IBM S370 and S390 interrupt processing is similar to that of the S360. In this section we discuss the way the PSW is treated when an interrupt occurs and when control returns to the interrupted program.

Recall, from Figure 8.1, that the instruction address is held in the PSW. The currently active PSW is shown in Figure 8.12 as Current PSW. Also shown are two sets of

Petitioner Exhibit 1053, Page 302 of 334

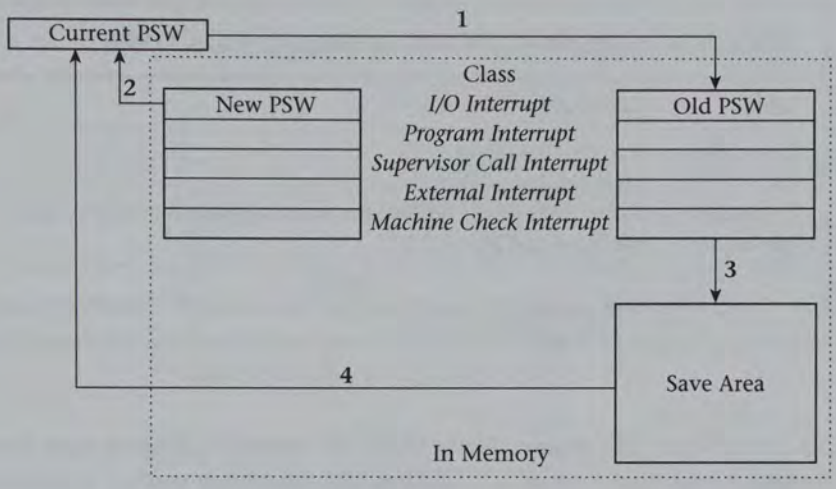


Figure 8.12 IBM S360 interrupt handling

five PSW memory locations, called Old PSW and New PSW. Each pair of PSWs is associated with the five classes of interrupts. An area in memory called Save Area is also shown.

When an interrupt occurs, its class is identified and the hardware moves the current PSW into the old PSW location, step 1, corresponding to the interrupt class. In addition, the hardware moves the corresponding new PSW, step 2, into the current PSW register. This PSW has been preconstructed by the operating system and will start the proper interrupt handler for the interrupt class. Step 3, also performed in hardware, moves the old PSW into the save area of memory, freeing up the old PSW location for another interrupt of the same class. When the interrupt handler completes its task it executes a LPSW instruction that returns the current PSW to its original state; step 2 or step 4.

Data Types. The IBM S360 interrupt data type is the PSW, shown in Figure 8.1. As noted above, the PSWs are different for the IBM S360, S370, and S390.

Addresses. The three areas shown in Figure 8.12 for the new PSW, the old PSW, and the save area are in real memory. Two addresses are needed: unmapped-memory address and an address implied in the op-code. Instructions that move PSWs as side effects also use unmapped-memory addresses.

Operations. The only explicit instruction for the support of interrupts is LPSW, which restores the state of the current PSW register, as shown in Table 8.8. Depending

TABLE 8.8 IBM S360 INTERRUPT SUPPORT INSTRUCTION

Name	Mnemonic (Format)	Function
Load PSW	LPSW (SI)	The current PSW is replaced with the double word $m(D_2 + (B_2))$. The current PSW is not retained

TABLE 8.9 INTEL PENTIUM PRO INTERRUPT DATA TYPES

Name	Description
EFLAGFS	Flag register, Figure 3.12
CS	Code segment selector*
EIP register	Instruction pointer holding the current program counter value
ESP register	Stack pointer, Figure 3.7

* The code segment registers hold the segment selectors that index into the descriptor table.

on the address, the load is either from the new PSW or save area in memory. Other nonprivileged instructions are used by the operating system to compose the various PSWs in memory.

Intel Pentium Pro

The Pentium uses a vectored interrupt. When an interrupt or exception occurs, processor state information is pushed onto a stack. The return from the interrupt pops the stack to restore the interrupted program.

Data Types. The data types, shown in Table 8.9, are associated with the processor state and are saved in response to an interrupt or exception and restored with the return.

Addresses. The registers holding these data types are addressed by implication when the interrupt or exception occurs, with the return from the interrupt or exception, or when the stack pointer is incremented or decremented as a side effect.

Operations. Some of the operations that load, store, or change the interrupt data types are listed in Table 8.10. This list is not complete; it serves only to illustrate the type of instructions provided.

TABLE 8.10 INTEL PENTIUM PRO INTERRUPT INSTRUCTIONS

Name	Mnemonic	Function
Call to interrupt procedure	INT	Calls an interrupt handler found with the destination effective address. Pushes EFAGS, CS, and EIP as side effects onto the stack
Return from procedure	RET	Pops the stack that contains the EFAGS, CS, and EIP as a side effect
Move ESP	MOV ESP	Loads the stack pointer from the effective address in real memory
Pop a value from the stack	POP	Pops the top of the stack into DS, ES, SS, FS, or GS register and increments the stack pointer

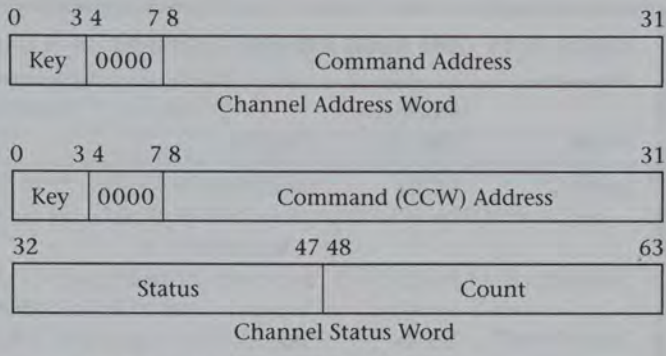


Figure 8.13 IBM S360 I/O data types

8.4 INPUT/OUTPUT SUPPORT

The general principles of I/O are described in Chapter 7. In this section the ISA support of I/O is described for two processors: the IBM S360 and the Intel Pentium Pro. The S360 is chosen because of its simplicity relative to the S370 and S390.

IBM S360

As described in Chapter 7, the IBM S360 I/O uses a coprocessor channel that executes its unique instruction set. The channels are programmable devices and can be described in typical ISA terms of data types, addresses, and operations (called channel command words). The study of the channel ISA is left to the student and is the subject of an exercise. The central processor initiates, stops, and monitors I/O activity with a small set of instructions having data types, addresses, and operations.

Data Types. Two data types are associated with I/O: the 32-bit channel address word (CAW) and the 64-bit channel status word (CSW) shown in Figure 8.13. The CAW is composed by the operating system and is stored in real memory starting at address 72. The command address in the CAW is the channel's starting program counter value. The 4-bit key is the memory protection key discussed previously in Section 8.1.

The CSW is maintained in real memory starting at address 64 and can be interrogated by the CPU to assess the status of the currently active channel. This double word contains the current channel command (CCW) address (the channel program counter value), various status bits, and a current count of I/O transfers. The 4-bit key is the memory protection key discussed previously in Section 8.1.

Addresses. The effective addresses of the I/O support instructions are the channel and the device addresses, not memory addresses. The 24-bit base plus displacement effective address is truncated to 16 bits: 8 bits address 256 channels and 8 bits address 256 devices per channel. The CAW and the CSW are addressed by implication.

Operations. There are four instructions for the IBM S360 I/O, as shown in Table 8.11. An I/O operation is initiated by the CPU by using the privileged instruction SIO. This instruction moves a program counter value from the CAW in real-memory location 72 into the program counter of the addressed channel, as shown in Figure 8.14. The channel starts executing its instructions stored in main memory. At any time, the CPU, using the privileged instruction TIO, loads status information from

Petitioner Exhibit 1053, Page 305 of 334

TABLE 8.11 IBM S360 I/O INSTRUCTIONS

Name	Mnemonic (Format)	Function
Start I/O*	SIO(S)	Sends CAW at location 72 to the channel addressed by $(D_1 + (B_1))$ and starts the channel executing its program
Test I/O*	TIO(S)	Sets condition code in the CPU PSW for status of channel and device at address $(D_1 + (B_1))$
Test channel*	TCH(S)	Sets condition code in the CPU PSW for channel status at address $(D_1 + (B_1))$
Halt I/O*	HIO(S)	Terminates channel and device operation

* Privileged.

the channel and the device into the PSW. The TCH instruction places channel status information into the current PSW. The HIO instruction stops the current executing channel program and the device data transfer.

Intel Pentium Pro

The Pentium has two I/O facilities: I/O ports and memory-mapped I/O. The Pentium Pro I/O has evolved over many generations of processors, all the while maintaining compatibility with earlier processors and programs.

Data Types. There are no data types, as such, associated with I/O. The I/O operations transfer bits among the registers, memory locations, and the ports. These bits are allocated to the specific I/O operation required by the system.

Addresses. The I/O space is a separate address space of 64K 8-bit ports numbered 0 through FFFH. Two consecutive 8-bit ports can be used as a 16-bit port and two consecutive 16-bit ports can be used as a 32-bit port. The I/O address space is shown in Figure 7.5.

Operations. A set of instructions provides for I/O operations by means of ports. However, the ports are allocated to the processor's memory-address lines. The I/O

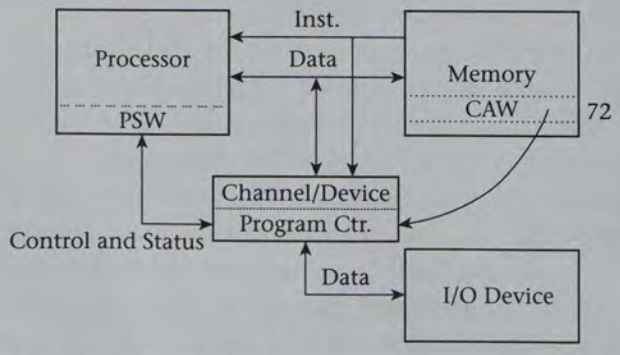


Figure 8.14 IBM S360 I/O operation

Petitioner Exhibit 1053, Page 306 of 334

TABLE 8.12 INTEL PENTIUM PRO I/O INSTRUCTIONS

Name	Mnemonic	Function
Input from port	IN	Inputs byte, word, or double word to AL, AX, or EAX register
Output to port	OUT	Outputs byte, word, or double word from AL, AX, or EAX register
Input from port to string*	INS	Inputs byte, word, or double word to ES register
Output string to port*	OUTS	Outputs byte, word, or double word from ES

* These instructions can be preceded by the REP (repeat) instruction for moving strings.

instructions indicate that the address lines are I/O addresses, not memory addresses.[78] Data are transmitted to/from the I/O device by means of the memory data lines. The I/O instructions with partial descriptions are shown in Table 8.12. These instructions are not privileged. Memory-mapped I/O instructions have been discussed in Chapter 7. However, note that any user move instruction can be used to transfer data to/from a memory-mapped I/O location.

8.5 CACHE SUPPORT

The original idea behind caches was that a cache is transparent to the user – a cache simply reduced the memory latency. With some modern processors, this simple view no longer applies. Direct intervention by the program is often useful to provide additional capabilities and improve the performance over a transparent interface to the memory. However, the transparent cache philosophy still holds for the IBM S390. The IBM S390 Principles of Operation, p. 2-2, state the following: "The effects, except on performance, of the physical construction and use of distinct storage media are not observable by the program" (IBM 1997). Thus, the IBM S360 family cannot be used to illustrate cache support extended instruction. Instead, the PowerPC will be used along with the Intel Pentium Pro to illustrate these features.

The events that follow a cache miss are viewed as side effects to a load or store instruction. The designers of some modern processors have found that this view can be expanded and provide direct program intervention of the cache. Program intervention is found in two areas:

1. direct invocation of the side effects, that is, making the side effect the primary operation of an instruction
2. setting the policies of the cache in order to have more flexibility than that provided by a wired-in default policy

[78] Processors before the Pentium Pro used an output pin to signal a memory or an I/O transaction.

Petitioner Exhibit 1053, Page 307 of 334

PowerPC

The PowerPC instruction and data cache support instructions are used as an example. Because of the functional difference between instruction and data caches, there are unique cache support instructions for both. For the details of these instructions, reference should be made to May et al. (1994).

Cache Side Effect Control. The PowerPC has six instructions that operate directly on the instruction or data caches. Two of these support instructions are provided for improving the performance of the memory system, as shown in Table 8.13. Four instructions support multiprocessor applications, discussed here rather than in Section 8.6, which discusses multiprocessor support. These instructions are used to enforce cache coherency in multiprocessor systems.

The data types are cache blocks and the addresses are translated real-memory addresses with their corresponding cache addresses.

TABLE 8.13 POWERPC CACHE SUPPORT INSTRUCTIONS

Name	Mnemonic	Function	Purpose
Data cache block touch	dcbt	Fetches a cache block associated with the virtual address ((RA) + (RB)) into the cache as would be done as a side effect for a miss on a dirty block	Prefetches a cache block before need. A subsequent load instruction using the address will have a cache hit.
Data cache block touch for store	dcbtst	Fetches a cache block associated with the virtual address ((RA) + (RB)) into the cache as would be done as a side effect for a miss on a dirty block	Prefetches a cache block before need. A subsequent store instruction using the address will have a cache hit. The block is in the cache; thus an allocate operation is not required for a store.
Data cache set to zero	dcbz	Sets all bytes in the block addressed ((RA) + (RB)) to zero.	Used to enforce cache coherency
Data cache block store	dcbst	Writes the block addressed by ((RA) + (RB)) to memory	Used to enforce cache coherency
Data cache block flush	dcbf	For the block addressed by ((RA) + (RB)); if unmodified, invalidate; if modified, copy to memory; if absent, invalidate other processors' caches	Used to enforce cache coherency
Instruction cache block invalidate	icbi	Invalidates the block addressed by ((RA) + (RB))	Used to enforce cache coherency
Instruction synchronization	isync	Waits until all previous instructions have been completed, then discards all prefetched instructions	Used to enforce context synchronization

Petitioner Exhibit 1053, Page 308 of 334

TABLE 8.14 POWERPC CACHE POLICY CONTROL

Bit	Purpose
W	Write through. A write must update higher levels of the cache, real memory, or I/O. Otherwise, write back.
I	Caching inhibited. Otherwise, references are cached.
M*	Memory coherence. All memory must be coherent. Otherwise, noncoherent.

* Useful for multiprocessor applications, discussed briefly in Section 4.3.

The cache support instructions are not privileged as they are used in user programs for performance enhancement and to enforce cache coherency in multiprocessor systems. The first two instructions in the table have as their primary purpose loading the cache before need, thereby improving system performance.

Cache Policy Control. Recall from Chapter 4 that cache organizations and policies are described that conform to the expected addressing pattern. However, it is possible for the operating system to dynamically adjust some cache policy factors to favor the expected short-term addressing patterns and use.

The data type consists of the policy or mode bits that are stored in page table entry. The three policy or mode control bits, W, I, and M, are in the page table entry data type. The use of these three bits is shown in Table 8.14. The policies are applied at the page level; thus different pages can operate under different policies. The cache addresses are translated real-memory addresses.

Operations. Because the mode control bits are allocated to the 128-bit page table entry, the only instructions associated with cache mode control are the instructions that load and store page tables in real memory. As the PowerPC is a load/store architecture, page tables are moved in memory by sequences of load and store instructions.

Intel Pentium Pro

The Pentium Pro can be used in multiprocessor configurations, and program optimization is desirable. Thus this processor has instructions for direct control of its cache.

Cache Side Effect Control. The Pentium Pro has two instructions that operate directly on the L1 and L2 caches and indirectly on the L3 cache, shown in Table 8.15. These instructions are privileged.

The term invalidate is used to signify that the valid bits are changed to not valid. Some literature uses the term purge or flush to signify the same action. Flush is confusing, as this term is used in pipelined processors to signify that the work in progress is flushed from the pipeline and is completed.

Cache Policy Control. Figure 8.4 shows the Pentium Pro CRs. In CR0, bit 30 (CD, cache disable) and bit 29 (NW, not write through) set the cache policy, as shown in Table 8.16. Because the policy bits are in CR0, the policy is global in scope. This is unlike the PowerPC, in which the cache policy is set on a page basis.

Petitioner Exhibit 1053, Page 309 of 334

TABLE 8.15 INTEL PENTIUM PRO CACHE SUPPORT INSTRUCTIONS

Name	Mnemonic	Function	Purpose
Invalidate internal caches	INVD	Invalidates the L1 and L2 caches and issues a special-function bus cycle to initiate external L3 cache invalidation. Data in the L1, L2, and L3 caches are not written back to real memory*	Permits the operating system to invalidate the cache when the system is rebooted or there is a context switch
Write back and invalidate cache	WBINVD	Writes back all modified L1 and L2 cache blocks to real memory and then invalidates the blocks. Issues a special function bus cycle to L3 caches to write back and invalidate	Permits the operating system to invalidate the cache when the system is rebooted or there is a context switch. The caches are written back to update all changes

* The L1 instruction and data caches are on the processor chip. The L2 unified cache is in the package and the L3 cache is outside the package; see Figure 4.10.

TABLE 8.16 INTEL PENTIUM PRO CACHE POLICY

CD	NW	Cache Policy
0	0	Write back
0	1	Write through
1	x	Cache disabled

Two bits in the control register control the caching policy for page tables. In CR3, bit 4 (PCD, page-level cache disable) and bit 3 (PWT, page-level writes transparent) set the policy. The policy set by these two bits is shown in Table 8.17.

8.6 MULTIPROCESSOR SUPPORT

Modern computers are a collection of cooperating autonomous processors that function asynchronously. A single processor with an autonomous coprocessor I/O channel is an example of a multiprocessor. Two or more programs executing on one processor require synchronization; however, the following discussion is in terms of two or more processors. Because these processors function asynchronously, synchronization between them is required for the programs to execute correctly. Synchronization may be required at two levels: (1) the circuit level when processors are provided with separate clocks, and (2) the program level to provide explicit synchronization by the programs.

Petitioner Exhibit 1053, Page 310 of 334

TABLE 8.17 INTEL PENTIUM PRO PAGE TABLE CACHE POLICY

PCD	PWT	Cache Policy
0	x	Page table not cached
1	0	Page table cached, write through
1	1	Page table cached, write back

Circuit synchronization is discussed in Chapter 7. Thus this section discusses only program level synchronization. Two or more cooperating programs need to be synchronized so that variables will not be used by a processor if the variable is being modified by another processor. The most primitive means by which synchronization is established is by means of flags, called semaphores, which provide an interlock between the programs (Dijkstra 1968).

Semaphores work in the following way. Consider two processors accessing common memory by means of a bus, as shown in Figure 8.15. In the memory there is a datum that each processor will access and modify from time to time. It is imperative that if processor A has loaded the datum and is modifying it, processor B must not access the datum before processor A has written the modified datum back into memory – otherwise, processor B will obtain an incorrect value for the datum.

Two examples of applications that require synchronization are not selling the same airplane seat to two or more people if requested at the same time and not giving a busy signal to two callers who dial the same number at the same time. These are unlikely events, but must be guaranteed not to happen.

As shown in Figure 8.15, a semaphore S is allocated to each datum or to each block of data or program to be protected. If processor A can indicate to processor B, by means of the semaphore, that the datum is unavailable, the erroneous operation described above cannot happen.

The state for a binary single-bit semaphore can be defined as

$S = 1$, the datum is in use
$S = 0$, the datum is not in use.[79]

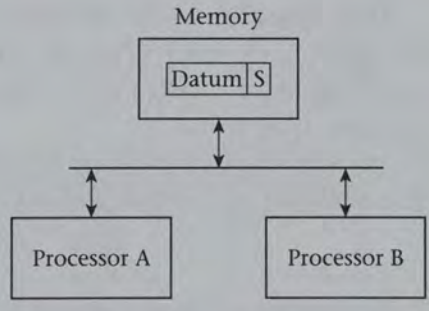


Figure 8.15 Multiprocessor synchronization

[79] Obviously, a system can be implemented that reverses this meaning of the semaphore.

Petitioner Exhibit 1053, Page 311 of 334

A processor accesses the semaphore and determines if it can use the datum or if the datum is in use by another process. Reading the semaphore must be done in such a way that any other processor cannot attempt to access the semaphore while this is going on. In other words, there must be exclusive access to the semaphore in real memory. A basic design problem in achieving this end is that the instruction must be able to read–modify–write to the memory location holding the semaphore; two data memory cycles are required.

One way of ensuring exclusive access to the semaphore is by means of an atomic instruction. An atomic instruction is an instruction that performs an operation that cannot be interrupted or terminated by another processor and the semaphore cannot be accessed by another processor during the execution of the instruction.

Two distinctly different techniques are used to achieve atomic synchronization instructions with the IBM S360 and the Intel x86 architecture (before the Pentium Pro). These are discussed in the two subsequent subsections.

IBM S360

One type of synchronization instruction is test and set, first used on the IBM S360 and included in the ISA of the S370 and S390 (Falkoff et al. 1964). This instruction is not privileged, is atomic, and performs the following steps[80]:

1. As a side effect, it locks out the memory bus from other users.
2. It copies the state of S into the CC bits of the PSW.
3. It performs a write that sets S to 1. If S is already 1, its value remains 1. (Collectively, steps 1, 2, and 3 are the test and set instruction. After this instruction is executed, the lock on the memory bus is released.)
4. The processor then tests the condition code with a branch instruction to determine if access to the datum has been obtained. If CC = 0, access is obtained, go to step 5. If CC = 1, access is not obtained; the processor tries again by returning to step 1.
5. The processor uses the protected datum, which may include modifying the datum, and goes to Step 6.
6. It executes an atomic store instruction that clears S to 0, thereby releasing the datum to another user, then continues.

Data Types. There are two data types associated with synchronization. The first is the semaphore, which is a byte rather than a single bit. This approach wastes 7 bits but simplifies the design as this architecture addresses to the byte; all the bits of the byte are either 1 or 0.

The second data type is the 2-bit CC field of the PSW shown in Figures 8.1 and 8.2. These bits are set to 00 for a semaphore of 0 and 01 for a semaphore of 1.

[80] IBM does not use the word atomic. Instead they use the phrase "appears to be an interlocked-update reference as observed by other CPUs." An I/O channel is not considered to be a CPU and interlocks are not provided.

Petitioner Exhibit 1053, Page 312 of 334

TABLE 8.18 IBM S360 PROCESSOR SYNCHRONIZATION INSTRUCTIONS

Name	Mnemonic (Format)	Function
Test and set	TS (SI)	The byte at address $(D_1 + (B_1))$ is copied into the condition code and 1 is written into all bits of the addressed byte
Move	MVI (SI)	Moves the immediate byte of all 0s in bits 8–15 into the byte at address $(D_1 + (B_1))$

Addresses. The effective address of the semaphore byte is base plus displacement. The other address for test and set, the CC, is implied in the op-code. The atomic store instruction uses an immediate operand of all zeros for the source. The destination address is the effective address of the semaphore byte.

Operations. Synchronization is provided by two atomic instructions, shown in Table 8.18. There are a number of complications in implementing the test and set instruction. For example, during a test and set instruction, the bus and memory are locked out from other users. As this instruction can take many clocks, the bus utilization can be adversely affected. An extreme case for the S370 and S390 would be that the first time the semaphore is accessed, it may be on the disk in virtual-address space, requiring a page fault, all the while locking out other processors from the memory system.

Other Multiprocessor Instructions. The IBM S360 provides a direct path for processors to communicate without using the system bus. Each processor has eight direct-in lines, eight direct-out lines, and eight direct-signal lines (bits 8–15 of the SI format instruction). Two instructions, shown in Table 8.19, use these lines for communication between other processors in the system.

There is a strobe signal with model-specific timing specifications that are not pertinent to the discussion of ISA.

TABLE 8.19 IBM S360 INTERPROCESSOR COMMUNICATION INSTRUCTIONS

Name	Mnemonic (Format)	Function
Read direct	RDD (SI)	Bits 8–15 of the instruction are placed on the signal-out lines. The byte on the direct-in lines is written into the byte at memory address $(D_1 + (B_1))$
Write direct	WRD (SI)	Bits 8–15 of the instruction are placed on the signal-out lines. The byte at memory address $(D_1 + (B_1))$ is placed on the direct-out lines

Petitioner Exhibit 1053, Page 313 of 334

TABLE 8.20 INTEL PENTIUM SYNCHRONIZATION INSTRUCTION

Name	Mnemonic	Function
Lock		Turns the accompanying instruction into an atomic instruction
Bit test and set	BTS	Stores the addressed bit in the CF flag and sets the addressed bit to 1
Bit test and reset	BTR	Stores the addressed bit in CF flag and resets the addressed bit to 0

Intel Pentium

The Intel Pentium performs the test and set instruction similarly to the IBM S360, except that an explicit LOCK prefix is added to the test and set instruction to lock the bus and memory rather than the lock's being asserted by side effect.[81] The lock is good for the one instruction with the LOCK prefix, after which the lock is released. The synchronization instruction, bit test set (BTS) transfers the semaphore value into the CF flag of the processor and sets the semaphore to 1. A test is then made on the CF flag to determine whether or not a valid access was obtained.

Data Types. There are two data types: the single-bit binary semaphore and the CF flag in the flag register, shown in Figure 8.4.

Addresses. Recall that the x86 architecture can address to the bit. This is accomplished by applying a mask to an addressed byte. Thus, unlike with the IBM S370, a semaphore can be a single bit. The CF flag that stores the semaphore needs no explicit address as its address is implied in the op-code.

Operations. The first byte of the bit test and set instruction contains the LOCK prefix (see Figure 3.15). There are a number of variations of the bit test and set instruction; however, only the BTS and the BTR instructions are shown in Table 8.20.

The Pentium provides several variations of the synchronization instruction that permit the meaning of the value in the semaphore to be different from the discussion above. That is, $S = 1$ means not in use and $S = 0$ means in use. Also, the two instructions, BTS and BTR, can be used to establish and clear synchronization when preceded by the LOCK prefix.

8.7 OTHER USER ISA ENHANCEMENTS, MMX

Over the years, a number of enhancements have been made to ISAs to support specific user applications. Examples of these enhancements are found in vector supercomputers, digital signal processing (DSP) microcomputers, and LISP machines for symbolic processing.

With the explosive growth in multimedia processing, ISA enhancements for this application are also being made. An example is found in the Intel MMX enhancements

[81] Seventeen instructions in the ISA can use the LOCK prefix. The Pentium Pro provides for automatic locking and the LOCK prefix is not required.

Petitioner Exhibit 1053, Page 314 of 334

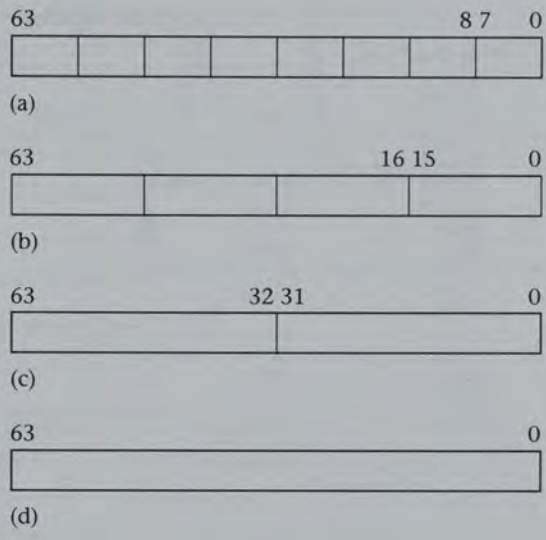


Figure 8.16 MMX data types (© 1996 IEEE)

to the Pentium Pro architecture (Intel 1997). The MMX enhancement is described with the same approach as used in the preceding subsections: data types, addresses, and operations.

Data Types. Intel designers noted that the main data types for multimedia algorithms are 8 bit pixels and 16-bit audio samples (Peleg and Weiser 1996). In addition, 32-bit and 64-bit integers are occasionally used. As the memory bus width of the Pentium is 64 bits, the memory-addressable unit is 64 bits and the other data types are allocated or packed into the AU. The MMX data types are shown in Figure 8.16. These data types are packed byte, packed word (16-bit words), packed double word, and quad word. Some of these data types are signed integers and others are ordinals.

Addresses. The designers needed to add MMX enhancements to the Pentium Pro ISA while maintaining upward compatibility with a minimum cost in additional chip area. Thus the basic memory architecture needed to be preserved. The MMX requirement led to a load/store register file architecture. In addition, with a need for a 64-bit data type, the integer registers of the Pentium Pro were inadequate. Thus the decision was made to use the eight 80-bit floating-point registers for MMX data. It was assumed that floating-point and MMX operations would not be needed at the same time.

For the type of processing envisioned for MMX, the floating-point registers presented the problem that they are organized as a push-down stack, discussed in Chapter 3, and MMX needed random access to operands. Thus the circuits of the floating-point registers were redesigned to give the registers dual-mode (random and stack) access capability. The 64-bit data types occupy the significand portion of the 80-bit internal IEEE format in the register file, as shown in Figure 8.17.

The exponent and the sign bits (bits 64–79) are set to all ones so that if, by error, a floating-point operation attempts to access the register file when occupied by MMX data, an interrupt error signal will be generated.

In addition to the register address, it is necessary to address memory for data loads and stores. These memory addresses are developed identically to all the other memory

Petitioner Exhibit 1053, Page 315 of 334

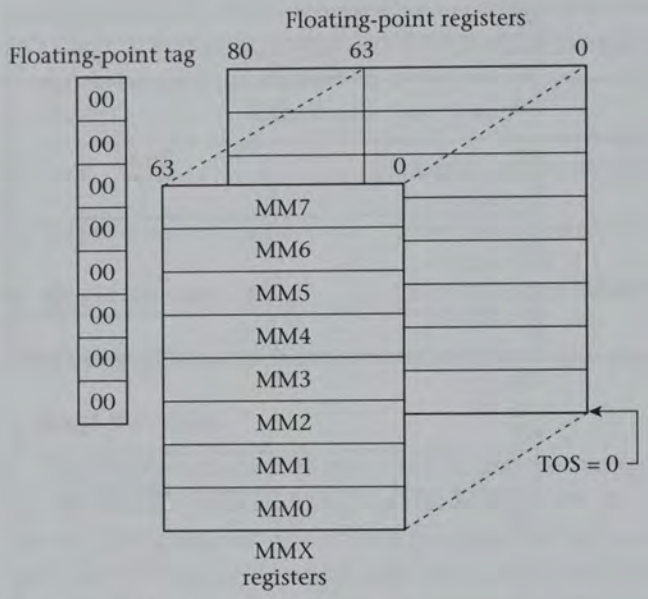


Figure 8.17 MMX register file (© 1996 IEEE)

addresses of the Pentium Pro ISA. That is, the effective address is mapped into a segmented paged virtual address. MMX data is cached in the data cache.

Operations. The MMX operations are added to the Pentium Pro ISA and consist of the basic arithmetic and logical operations applied to the packed data types. All control, such as branching, is handled normally with the Pentium Pro ISA. No indicator bits, or flags, were added. The Pentium Pro operational instructions have two addresses with the following options:

Ri ← Ri op Rj (two-address register)
Ri ← Ri op M(EA) (multiple accumulator)

The MMX modification provided a 64-bit ALU. The ALU can be partitioned, under the control of the instruction decoder, into two 32-bit ALUs, four 16-bit ALUs, or eight 8-bit ALUs. Because many of the multimedia data types are 8 and 16 bits, 8 or 4 integer operations can be performed by one instruction in one cycle. This subdivision of the ALU was pioneered with the ILLIAC-IV in the 1960s and was called single instruction multiple data (SIMD) by Flynn (Flynn 1966).

Note that for each of these data types, the total contents of the 64-bit AU is processed: 8 bytes, 4 words, 2 double words, or 1 quad word. As the processor is pipelined, when for example bytes are being added, the performance is 8 operations per clock while CPI = 1. The CPI measure has little relevance when measuring MMX operational performance.

EXAMPLE

If CPI is not a good measure of MMX performance, what could be?

Solution

In Chapter 2, the performance measure MFLOPS is used with supercomputers to measure the performance when executing floating-point operations. Let us

Petitioner Exhibit 1053, Page 316 of 334

suggest the following performance measure: millions of integer operations per second (MIOPS).

Comments

For a 300-MHz MMX processor executing 8-bit integer register-to-register operations, the peak MIOPS is

$$\text{MIOPS} = \frac{\text{clock frequency}}{10^6} \times \frac{64}{\text{number of operand bits}}$$

$$= \frac{300 \times 10^6}{10^6} \times \frac{64}{8} = 300 \times 8 = 2400.$$

Image and signal processing use an addition and subtraction mode called saturation arithmetic. This arithmetic provides that if the result of an addition or subtraction is greater or less than the expressible values, the domain function sets the result to the largest or the smallest representable value. The overflow or underflow indication is not saved in a flag. The MMX uses wraparound arithmetic when not using saturation arithmetic. Wraparound arithmetic truncates the result to the representable value. The examples below show addition and subtraction of the 8-bit ordinals 109 and 202.

	Addition	*Subtraction*
109	01101101	01101101
202	+11001010	−11001010
	100110111	110100011
Saturation Result	11111111	00000000
Wraparound Result	00110111	10100011

Table 8.21 shows the 47 MMX architected instructions. The MMX ISA has options for register-to-register operations and multiple accumulator operations. Note that with two exceptions, the instruction mnemonics start with the letter P for packed. Some of the data types are signed, others are ordinal (s,o); some of the arithmetic is wraparound while some is saturation (w,s).

To give a flavor of how the MMX instructions operate, three interesting instructions are now described. The first, as shown in Figure 8.18, is packed compare packed for greater than (PCMPGT) with 16-bit ordinal operands. With two 64-bit AUs in the register file, each of the words in one register is compared with the corresponding word in the other register. The words of one register are overwritten with ones if the word in the first register is greater than the word in the second register. Otherwise, the word is overwritten with zeros. The second operand can be in memory for a multiple accumulator operation. Variations of this instruction can be applied to bytes and doubles.

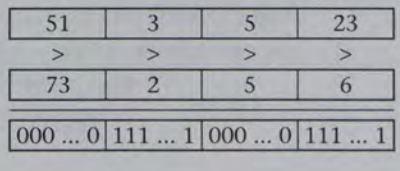


Figure 8.18 Packed compare instruction (© 1996 IEEE)

Petitioner Exhibit 1053, Page 317 of 334

TABLE 8.21 MMX INSTRUCTIONS

Operation	Options	Data Type Length	Signed, Ordinal	Wraparound Saturation	Total Number
PADD, PSUB		b, w, dw	s,o	w	12
PADDS, PSUBS		b, w	s	s	2
PADDUS, PSUBUS		b, w	o	s	2
PCMPE, PCMPGT	=, >	b, w, dw	o	w	6
PMULH, PMULL	Result, either high or low	w	s	w	2
PMADDWD		w	s	w	1
PUNPCKH, PUNPCKL	High or low	b, w, dw	s	s	6
PACKSS, PACKUS		w	s,o	s	3
PAND, PANDN	AND, NOT	qw	o	w	2
PSLL, PSRL	Logical left, right shift	w, dw, qw	o	w	6
PSRA	Arithmetic right shift	w, dw	s	w	2
MOVD, MOVQ	Mem → Reg Reg → Reg Reg → Mem	d, q			2
EMMS	Clears stack	Reg. tags			1
				TOTAL	47

The packed multiply and add (PMADDWD) instruction performs the multiply and add operation so important to signal processing applications (see Figure 3.18, which shows a matrix multiplication program). This MMX instruction, shown in Figure 8.19, has as its source operands two packed 16-bit signed words. It multiplies the words together producing four 32-bit products held in a temporary implemented register. These are then added and packed into two 32-bit packed double words that replace one of the source operands. The second packed operand can be in memory for processing in a multiple accumulator architecture.

EXAMPLE

What is the peak MIOPS for the PMADDWD instruction? The latency of this instruction is three clocks but it is pipelined, and one new pair of results is produced each clock. Assume a 300-MHz clock.

Solution

The PMADDWD instruction executes four multiplies and two adds per clock or six operations. Thus,

$$\text{MIOPS} = \frac{300 \times 10^6}{10^6} \times 6 = 1800.$$

Petitioner Exhibit 1053, Page 318 of 334

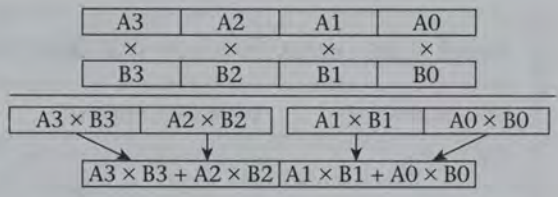


Figure 8.19 Packed multiply and add instruction (© 1996 IEEE)

The third instruction we consider is the unpack low packed data (PUNPCKLBW). This instruction can unpack bytes, words, and double words. Figure 8.20 shows the unpack bytes case. This instruction takes the four lower-order bytes of the right-hand word, stored in the register file, and interdigitates the bytes from the other source (which may be in a register or in memory). The result is stored in the first operand source register.

Packing and unpacking instructions are required for preparing data for processing in the SIMD style of the MMX instruction set architecture. These instructions are PACKSS, PACKUS, PUNPCKH, and PUNKPCKL.

8.7.1 OTHER INTEL ISA SUPPORT

The Pentium Pro provides special ISA support for software development. Each model of the Pentium family is provided with model-specific registers (MSRs) that store the support data types. This ISA support is useful for such tasks as debugging programs, system tuning, and monitoring cache miss rates.

Several dedicated registers are architected, namely, debug registers, a performance monitoring register, and a time stamp counter. Unique instructions are provided in the ISA for establishing the function of these registers and for loading and storing them. Details can be found in the Intel Pentium Pro User's Manual (Intel Corporation 1996). However, here we discuss the performance monitoring register.

The Pentium Pro has two 40-bit performance counters that can simultaneously monitor various events internal to the processor. The counters can tally the number of events or the duration of an event measured in clocks. Two 32-bit registers set up the particular function the performance counters are to monitor, and privileged instructions configure the bits in these registers.

An extensive list of 68 events can be monitored by these counters. A few of these are listed in Table 8.22. The complete list can be found in the Intel documentation. From this short list of the Pentium Pro performance monitoring capability, one can see that many of the parameters needed for performance modeling, discussed in Chapter 2, can be obtained with the processor hardware. These parameters in turn can be used to tune programs and to find improvements in the hardware implementation that may be beneficial for future models of the processor.

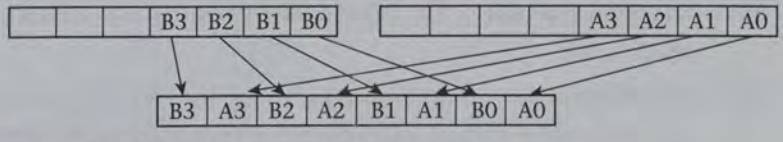


Figure 8.20 Unpack byte data (© 1996 IEEE)

Petitioner Exhibit 1053, Page 319 of 334

TABLE 8.22 INTEL PENTIUM PRO PERFORMANCE MONITOR EVENTS

Unit Monitored	Event Description
Data cache	• All memory references • Weighted number of cycles while a data cache miss is outstanding
Instruction fetch unit	• Number of instruction fetches • Number of instruction TLB misses • Number of instruction fetch misses
L2 cache	• Number of L2 instruction fetches • Number of L2 data loads • Number of L2 data stores
Floating-point unit	• Number of computational floating-point operations retired • Number of computational floating-point operations executed

Note that these performance monitoring facilities are similar to those found with the IBM S370. Table 8.1 shows that words 9, 10, and 11 are allocated to recording program events.

REFERENCES

Dijkstra, E. W. (1968). "Co-Operating Sequential Processes," in *Programming Languages*, F. deGenuys, ed., Academic, New York.

Falkoff, J. W., Iverson, K. E., and Sussenguth, E. H. (1964). "A Formal Description of System /360," *IBM Systems Journal*, **3**:3.

Flynn, M. J. (1966). "Very High-Speed Computing Systems," *Proceedings of the IEEE*, **54**, pp. 1901–1990.

IBM (1970). "IBM S/360 Principles of Operation," order GA22-6821-8.

IBM (1976). "IBM System/370 Principles of Operation," order GA22-7000-5.

IBM (1997). "Enterprise System Architecture/390 Principles of Operation," order SA- 7201-04.

Intel Corporation (1993). "Pentium Processor User's Manual," Vol. 3, order 241-430-001.

Intel Corporation (1996). "Pentium Pro Family Developer's Manual," Vols. 1–3, order 000901-001.

Intel Corporation (1997). "Intel Architecture MMX™ Technology, Programmer's Reference Manual," order 243007-003.

May, C., Silha, E., Simpson, R., and Warren, H., eds. (1994). *The PowerPC Architecture*, Morgan Kaufmann, San Francisco, CA.

Peleg, A. and Weiser, U. (1996). "MMX Technology Extension to the Intel Architecture," *IEEE Micro*, **16**:4, pp. 42–50.

EXERCISES

8.1 Describe the IBM S360/370 channel ISA in terms of data types, addresses, and operations.

8.2 What instruction could be used with the IBM S360 to determine if the condition code permits access to a datum that is protected with a semaphore?

Petitioner Exhibit 1053, Page 320 of 334

8.3 Consult the Intel Pentium Pro documentation on performance monitoring and determine how the miss rate of the instruction cache and the data cache can be determined.

8.4 The paper "PicoJava: A Direct Execution Engine for Java Bytecode" by McGhan and O'Connor, found in *Computer*, October 1998, describes a processor for executing Java programs. Write an essay describing the salient architecture and implementation features of this processor in 500 words or less.

8.5 Describe the meaning of the circled numbers 1, 2, and 3, of Figure 8.6.

8.6 Section 4.6 describes four segment/page configurations that are used with the IBM S370. For each of these configurations, list the number of bits that are added in the two adders and the bits concatenated in the real-address register of Figure 8.6. Recall that the S370 has a 24-bit effective or logical address.

8.7 What is the length of the virtual address of the Pentium Pro?

8.8 Figure 8.8 shows the Pentium Pro segmented virtual-address system for 4K-byte pages. The system can also support 1M-byte pages. Show how the linear address and tables are reconfigured for 1M-byte pages.

8.9 Explain why two adders are used for the S370 virtual-address translation, Figure 8.6, whereas only one adder is used for the Pentium Pro, Figure 8.8.

8.10 Consider the synchronization instructions discussed in Section 8.6. What happens when two or more requests to access the semaphore arrive on exactly the same clock?

8.11 Derive the model for MIOPS used in the example in Section 8.7.

8.12 What is done with the two 32-bit results computed in the multiply and add instruction shown in Figure 8.19? In other words, how can these results be used as inputs to subsequent instructions? Note that the source operands are 16 bits.

Petitioner Exhibit 1053, Page 321 of 334

INDEX

access time, 29, 47, 163, 168, 170, 171, 239, 251
accumulator, 6, 8, 20, 21, 54, 56, 57, 66, 90, 98, 99, 106, 202, 243, 244, 304, 305, 306
  multiple, 57, 305, 306
adder, 16, 77, 110, 226, 309
Addition/Subtraction, 111, 118
address, 2, 6, 7, 13, 24, 50, 52, 53–60, 61, 64, 66, 67, 69–71, 88, 89, 92, 93, 97, 98, 99, 101–3, 133–40, 141, 143–5, 148, 149–51, 162, 163, 166, 167, 170–7, 178, 181, 182, 184–6, 187, 194, 197, 202, 204, 205, 212–5, 217, 219, 221–3, 225, 226, 231, 232, 237, 241, 244–6, 249, 253, 278, 281, 284, 285, 287–9, 291–5, 297, 301–4, 309
  alignment, 52, 58, 82, 84, 235, 239
  effective, 2, 60, 61, 66, 89, 98, 99, 133, 174, 182, 186, 195, 204, 205, 212–14, 221, 225, 226, 278, 284, 293, 301, 304
  explicit, 52, 54, 56, 62, 63, 65, 74, 81, 85, 106, 118, 232, 302
  implicit, 54, 63, 74, 85, 86, 259, 281
  logical, 60, 174, 175
  MMX, 303
  one address accumulator, 56
  three address, memory to memory, 55
  three address, register to register, 55
  translation, dynamic (S370), 284
  two address, memory to memory, 55
  zero address or stack, 56
Addressable Unit (AU), 58, 59, 64, 93, 131, 134–6, 138, 149–53, 158, 164, 166, 167, 173, 174, 177, 303–5
Aiken, Howard, 2, 14, 19, 20, 23
allocate, 88, 94, 98, 128, 180
Amdahl's law, 37, 107
architected, 13, 14, 50, 62, 64–6, 73, 77, 81, 88, 90, 91, 93, 102, 114, 117, 118, 120, 184, 195, 242, 305, 307
Architectural Styles, 98
architecture, 2, 3, 5, 7, 10, 13, 18, 42, 51, 55, 56, 57, 60, 62, 64, 65, 80, 95, 98, 99, 101, 102, 105, 106, 182, 183, 186, 233, 234, 236, 237, 297, 300, 302, 303, 306, 307, 309
arithmetic, 2, 5, 6, 13–16, 19, 20, 21, 24, 29–32, 35, 36, 38, 47, 49, 50, 62, 65, 70, 71, 75–7, 79–82, 84–7, 90, 91, 99, 104, 106, 109, 110, 111, 112, 113, 114, 116–17, 118, 120–3, 125–7, 128, 130, 143–5, 160, 163, 205, 219, 224, 232, 237, 303–6
  floating point, 118
  saturation, 305
Arithmetic and Logic Unit (ALU), 6, 8, 10, 76, 78, 86, 90, 102, 122, 183–5, 191–3, 195–200, 225, 230, 237, 238, 240, 304
Atanasoff Berry Computer (ABC), 15, 16, 17, 23, 25, 130
Atanasoff, John, 14–16, 19, 24, 77, 130

Babbage, Charles, 2, 14, 15, 16, 19, 20, 24, 127, 128
bandwidth, 5, 47, 63, 132, 133, 164, 167–9, 170, 171, 241, 248, 249, 251, 252, 253, 257–9
base, 2, 16, 17, 25, 82, 84, 85, 87, 88, 122, 130
Benchly Park, 2
benchmark, 35, 219, 220
  SPEC, 127
  SPICE, 46
biased, 35, 80–4, 86, 118–21, 128, 129
binary, 16, 25, 75–8, 80, 85, 88, 110, 117, 118, 120, 121, 129, 130, 177, 178, 260, 262, 299, 302
binding, 38, 51, 52, 94, 129
bit budget, 57, 58, 94, 96, 100, 103–4, 105
bit stuffing, 262
bit traffic, 100, 103–5, 109
bit vector, 88, 173
block diagram, 7, 17, 141, 159, 160, 221, 223
branch strategy, 212
  predict not take, 215
  predict take, 214

Petitioner Exhibit 1053, Page 322 of 334

Branch Target Buffer (BTB), 216, 221–4, 239, 240
Branch Target Cache (BTC), 221–4
branch, 8, 64, 67, 90–2, 96, 102, 108, 183, 192, 196, 199, 212, 213, 217–19, 221, 222, 224, 225, 300
  conditional, 8, 18, 64, 65, 67, 92, 100, 128, 196, 197, 212, 218, 219, 221
  delay, Weighted Average Branch Delay (WABD), 212, 213, 215, 217, 218, 221–4, 239, 240
  delayed, 224
  pipeline stages defined, 212
  prediction, branch history, 219
  prediction, instruction characteristics, 217
  statistics, 219
  strategy, 212, 213, 215, 217, 218, 224, 239, 240
  strategy, general prediction, 216
  strategy, pipeline freeze, 212
  target, 51, 64, 67, 187, 196, 197, 202, 212–17, 219, 221–4, 239, 240
  unconditional, 8, 64, 65, 66, 67, 128, 196, 197, 199, 218
buffer, 144, 161, 167, 170, 180, 185, 195, 213, 215, 216, 221, 224, 230, 239, 240
bus, 21, 60, 90, 100, 103, 128, 159, 161, 164–7, 182, 185, 196, 206, 226, 244, 248, 249, 252–8, 264, 273, 274, 299, 300–3
  asynchronous communications, 256, 257
  centralized control, 254, 255
  clocking and deskewing, 257, 274
  decentralized control, 253, 255
  dedicated, 252, 253, 258
  design examples, 258
  general purpose, 252, 253
  synchronous communications, 257
  utilization, 127, 301
bypassing, 228
bytecode, 309

C++, 106
cache, 145
  address, 148, 149, 171, 297
  allocation and organization, 148
  associative, 150, 151
  block size, 158, 165, 166, 182
  block slot, 145, 149–53
  canonical, 149
  coherency, 227, 238, 248, 296, 297
  control of policy, 297
  control of side effects, 296, 297
  data, 156
  direct, 6, 7, 42, 50, 52, 65, 93, 98, 136–9, 142, 144, 148, 150, 151, 156, 157, 172, 182, 197, 222, 231, 240, 279, 284, 295, 297, 301
  disk, SMARTDrive, 179
  disk, VCACHE, 179
  global hit probabilities, 160
  hit, 153, 182, 193
  instruction, 155
  line, 181
  local miss probabilities, 160
  miss, 134, 148, 152, 154, 155, 157, 159, 162, 172, 181, 187, 226, 227, 239, 240, 295, 307, 309
  model, 160
  multilevel, 159
  organization, 152, 297
  policy control, 297
  read and write policies, 153
  second level, 157, 159, 161, 162, 225
  sector, 149, 150, 152, 177, 178
  set, 149, 151, 152, 156, 173
  set associative, 150, 151, 152, 156, 157, 161, 163, 165, 172, 182, 222, 224
  set conflict miss, 151, 224
  set index, 149, 150
  side effect control, 296, 297
  split, 147, 227, 237
  support instructions, 295
  topology, 147
  unified, 147, 157, 158, 160, 161, 165, 180, 225, 226, 227, 239, 240
  valid bit, 149, 150, 151, 153, 182, 289, 297
  virtual and real address, 171
  write direct, 154
  write through, 153, 154, 180, 226
call/return, 70
Cambridge University, 66, 243
CDC 6600, 74, 84, 102, 116, 248
CDC 7600, 74
CDROM, 250
Central Processing Unit (CPU), 3, 5, 181, 278, 279, 293, 300
central tendency, 29–36
Channel Address Word (CAW), 293
Channel Command Address (CCW), 293
Channel Status Word (CSW), 293
Chrysler Corp., 1
CISC, 51, 57, 98–101, 106, 109, 127, 201, 244
clock, 26, 33, 44, 46, 49, 91, 109, 155, 157, 164, 166, 171, 172, 185, 186, 188, 190, 193, 197, 199, 200, 202, 205, 207–11, 214, 217, 222, 224–9, 234, 237, 238, 240, 242, 256, 257, 259–65, 274, 304, 309
clock, distribution, 208
clock, frequency, 44, 46, 109
clock, period, 26, 33, 157, 164, 207, 208, 210, 214, 217, 222, 225, 257, 264, 265
clock, recovery, 259, 260, 263
  Manchester code, 260
  NRZ, 259
clock, skew, 208, 209, 257, 259, 274

Petitioner Exhibit 1053, Page 323 of 334

clock, synchronization, 260
  metastable, 263–5
  synchronizer, 264, 265
coherency, 227, 238, 248, 296, 297
collision, 140–2
Colossus, 2, 23, 24
Common Data bus (CDB), 226
compiler, 4, 51, 61–3, 106, 108, 217, 225, 228
compound growth rate, 37
concurrency, 5, 18, 26, 185, 200, 201, 225, 237, 243
condition codes, 91
contest switch, 4, 53, 63, 67, 88, 90, 91, 135, 139, 154, 163, 221, 277, 283, 290
control, hardwired, 185
  microprogrammed, 193
counter, global, 160, 254

daisy chain, 254, 255
data flow, 3, 21
data path, 184–6, 188, 192, 193–6, 200, 201, 243, 258
data type, 50
  Boolean, 62
  control register (S370), 278, 29–8, 301, 308
  control registers (Pentium Pro), 281, 287
  excess three, 77, 78, 80, 82
  flag registers (Pentium Pro), 281, 302
  floating point, 75, 80, 87, 115, 118, 126
  fraction, 7, 27, 29, 30, 32, 35, 38, 39, 43, 75, 80–8, 95, 105, 111–13, 115, 117, 118, 134, 146, 147, 160, 161, 218, 240, 263, 278
  immediate or literal, 54, 64, 93, 96, 102, 128, 129, 244, 301
  integer, 52, 57, 59, 60, 62, 74–6, 93, 101, 103, 114, 117, 135, 303
  mixed number, 75, 80, 81, 85, 92, 118, 119, 243
  MMX, 303
  one's complement, 16, 76, 77, 81, 111, 112, 117, 120
  packed, 303, 305, 306, 307
  page directory (x86), 138, 176, 281, 286, 288, 289, 290
  Program Status Word (PSW), 284, 287, 288, 291, 293, 299, 303, 305, 306, 309
  segment descriptor (x86), 175, 176, 286, 287, 290
  segment selector (x86), 176, 286, 290
  segment table entry (S370), 175, 182, 284, 285
  segment, descriptor (x86), 175, 176, 286, 287, 290
  sign and magnitude, 20, 76, 77, 80, 81, 84, 87, 111, 117, 126
  signed decimal, 80
  table entry, (S370), 284, 285
  TLB entry (S370), 284, 290
  two's complement, 7, 76, 77, 81, 84, 101, 103, 111, 113, 117
  two out of five code, 77
  unsigned decimal, 77
  unsigned integer, 75
  user, 6, 50–2, 58, 59, 63, 74–7, 80–90, 93, 95, 101, 103, 110–15, 117, 119–21, 123, 126, 129, 130, 131
DEC VAX, 106, 109, 202
decimal adjust, 80, 129
decode, 11, 187, 195, 204, 205, 234
dependencies, 22, 211, 227, 228, 230, 231, 237, 239, 240
diagnose, 264
Difference Engine, 2, 14, 15, 24
differences, 14, 15, 19, 98
digit, 14, 20, 21, 76–80, 85, 110, 122, 129, 140, 142, 182, 244, 260
disk, 4, 40, 41, 45, 47, 132, 133–7, 143, 144, 148, 154, 173, 176–80, 241, 243, 246, 247, 250–4, 263, 266, 269, 279, 301
  cache, 144, 178–80
  latency, 135, 177, 247
  seek time, 177
  zoned, 177, 181
displacement, 55, 89, 98, 135–8, 150, 151, 171–3, 183, 185, 195, 219, 293, 301
division, 16, 20, 62, 102, 114–9, 123, 135, 208, 210
  non restoring, 115, 116, 117
  restoring, 16, 69, 71, 72, 115–17
domain function, 62, 115, 12–14, 126, 130, 305
DRAMs, 6, 16, 43, 44, 47, 54, 164, 167–71, 177

early select, 135, 150
Eckert, J. Presper, 14, 21, 65, 127
EDSAC, 66, 243, 244
Endian, Big, 59, 60
Endian, Little, 59, 60
ENIAC, 2, 16, 21–5, 77
evict, 135, 152–4, 224
exceptions, 71, 156, 292
Excess Low-order Bits (ELBs), 121–5
execute, 11, 195, 202, 204, 234
execution unit, 46, 226, 228–30, 234
expanded op-codes, 50, 95, 96, 128, 190, 192, 197, 278, 295
exponent, 80–4, 86–88, 118–22, 126, 129–30, 303
  biased, 80–3, 120, 128
extended instruction set architectures, 275

Fairchild Semiconductor, 39
firing tables, 21
flags, 4, 52, 91, 92, 93, 100, 299, 304
Ford, 1, 2, 204
FORTRAN, 13

Petitioner Exhibit 1053, Page 324 of 334

forwarding, 228, 229, 237, 239, 240
frequency modulation, 259

garbage collection, 173, 176
Gaussian elimination, 16, 130
General Motors, 1
global hit rate, 160, 254, 287
guard bits, 121, 123
guard digit, 123

Harvard Architecture, 2
Harvard University, 2, 19, 20, 203
hashing, 140, 142, 163, 181, 182
hazards, 211
  data, read after write, 277
  data write after read, 227
  data write after write, 227, 230
  other spaces, 231
  structural, 225–7, 237, 238, 265
Hewlett Packard, 139
hidden bit, 86, 129
hit ratio, 223

IBM 650, 77
IBM 701, 13, 244, 246
IBM 704, 13, 82, 90
IBM 709, 248
IBM PowerPC, 24, 58, 60, 94, 96, 123, 138, 139, 156, 217, 231, 246, 29–7, 308
IBM RS/6000, 139
IBM S360, 57, 58, 60, 64, 66, 72, 77, 81–4, 88, 90, 91, 94, 98, 101, 105, 106, 108, 122, 123, 129–31, 146, 173, 194, 200, 201, 202, 226, 229, 246, 248, 275, 277–9, 283, 290, 291, 293, 295, 300–2, 308
IBM S370, 57, 77, 88, 138, 163, 174, 182, 277–9, 283–5, 291, 293, 300–2, 308, 309
IBM S390, 57, 60, 77, 88, 130, 205, 276, 277–9, 291, 293, 295, 300, 301
IEEE floating point standard, 85, 101
immediate operand, 54, 96, 128, 244, 301
implementation, 5, 7, 13, 64, 66, 81, 100, 114, 120, 123, 131, 156, 191, 192, 199, 200, 202, 211, 309
independent request, 254–6
indexing, 6, 12, 13, 18, 50, 53, 60, 140
indicator bits, 42, 52, 64, 67, 69, 88, 90, 91, 93, 102, 108, 276, 281, 304
infinite series, 19
Input/Output, 3, 5, 7, 146, 161, 226, 241, 293
  address, 294, 295
  channel, 248, 251, 263, 298, 300
  coprocessor, 123, 242, 248, 252, 293, 298
  device requirements, 249
  Direct Memory Access (DMA), 5, 248, 249
  memory mapped, 244, 294
  programmed, 242, 243, 248
  support, 293
  system architecture, 241
Institute for Advanced Study, 5, 6
instruction, 57
  address, 88, 278
  atomic, 108, 300, 301
  composition, 93
  dyadic, 82, 84, 86, 87, 91, 93, 94, 98, 106, 122, 130
  fetch, 10, 11, 101, 146, 147, 187, 193, 194, 195, 199, 202, 205, 212, 234, 237
  format, S370, 66, 279
  format, MIPS R2000, 99
  format Pentium Pro, 88, 101
  format, PowerPC, 96
  monadic, 82, 84, 86, 87, 91, 93, 94, 98, 106, 122, 130
Instruction Buffer Register (IBR), 10, 11
Instruction Level Parallelism (ILP), 236
Instruction Pointer Register (EIP), 88, 277, 280, 281, 283
Instruction Set Architecture (ISA), 5, 6, 8, 12–14, 20, 24, 50–4, 56, 57, 62–5, 69, 70, 72, 74, 77, 88, 90, 93, 96, 98, 102, 105, 109, 110, 128, 129, 139, 146, 183, 186, 190, 193–200, 202–4, 239, 242–4, 248, 274, 275, 283, 293, 300, 302–5, 307, 308
instructions, 2–8, 10–13, 18, 20, 27, 28, 30, 32, 45, 46, 50–8, 60–7, 71, 72, 82, 90, 92, 94–102, 104–9, 114, 120, 128,
instructions, architected, 305
  control, 64, 275
  move, 63
  privileged, 275, 276, 277, 279, 285, 292, 307
  protected mode, 275
  semiprivileged, 276, 279
Instructions Per Clock (IPC), 27, 205
integrated circuit, 40, 42, 47
Intel 8008, 78, 95, 99, 279
Intel 8048, 244
Intel 8080, 95, 123, 244, 274
Intel 8086, 101, 108, 127, 130
Intel i286, 91, 99, 176
Intel i386, 176
Intel i486, 99, 173, 181
Intel x86, 7, 42, 58, 60, 64, 78, 80, 91, 92, 94–6, 98–03, 105, 106, 108, 129–31, 300, 302
Intel Pentium, 24, 44–6, 57, 60, 67–9, 88, 90–2, 99–101, 128, 130, 138, 139, 147, 156, 161, 163, 175, 182, 205, 210, 231, 244–7, 275, 276, 279, 280, 281, 286, 289, 290, 292–5, 297, 300, 302–4, 307–9
Intel Pentium Pro MMX extensions, 99, 101, 128, 276, 302, 303, 304, 305, 306, 307, 308
interlock, 299, 300
Internet, 241, 250, 266, 267, 269, 270, 271
interpretation, 7, 10, 12, 50, 51, 121, 183, 186, 204, 207

Petitioner Exhibit 1053, Page 325 of 334

interpreters, 4
interrupt, 64, 69–74, 89, 92, 126, 202, 211, 231–3, 239, 246, 247, 254, 275, 276, 278, 283, 285, 290–2
  asynchronous, 70
  continuation, 72
  exception, 64, 69, 70, 90, 305
  following instructions, 233
  handler, 71, 72, 73, 92, 211, 233, 291
  I/O, 246, 247
  identification of source, 71
  imprecise, 232
  internal, 70, 181, 232
  interrupted stage, 232
  interrupting stage, 232, 239
  invalidate, 297
  preceding instructions, 233
  precise, 121, 123, 125, 130, 231, 233, 239
  restart, 72
  support, 290
  synchronous, 70
  vectored, 72, 292
Iowa State College, University, 2, 15, 17

Java, 309
jump, 53, 65, 77, 96

key, 5, 6, 95, 143, 278, 293
key in storage, 278

language, 50, 51, 53, 62, 63, 105, 106, 183, 195, 198
latch, 208, 209
latency, 4, 5, 29, 72, 132, 133, 135, 137, 138, 142–7, 154, 155, 157, 159, 160, 162, 163–68, 177, 178, 182, 204, 211, 227, 241, 243, 246, 247, 249, 251–3, 295, 306
learning curve, 42–5, 49, 158
Least Recently Used (LRU), 153
Least Significant Bit (LSB), 10, 59, 75, 92, 94, 103, 114, 122–6, 129, 151, 152, 166, 167
lexical level, 51
linear equations, 14, 15, 17, 18
list, linked, 21, 53, 69, 72, 130, 140, 141, 173, 194–6, 249, 250, 252, 274, 307, 309
load/store, 54, 55, 57, 63, 98, 102, 146, 183, 186, 204, 236, 237, 297, 303
load true dependencies, 237
locality, 4, 134, 135
  spatial, 134, 135, 155, 156
  temporal, 134, 135, 139, 156, 161, 221
logical, 16, 24, 60, 62, 74, 75, 88, 92, 114, 174, 175, 225, 229, 246, 285, 304, 309
London Science Museum, 273
look ahead, 135, 156, 158
look behind, 135, 156, 158
Lotus 1–2–3, 176

macro program counter, 194
magnetic disk, 132, 177, 244, 267
mainframe, 1, 29, 46, 250, 251, 252
Manchester University, 13, 60, 202, 260
mantissa, 80–2, 85, 86, 118
mapped, 149, 231, 244–7, 279, 295, 304
mapping, congruence, 135
Mark I, 13, 20, 21, 23
mask, 278, 279
matrix, 60, 61, 104, 105, 306
Mauchly, John, 14, 21
MC68000, 72, 123
MC88110, 237
Mean Time Between Failure (MTBF), 264
mean, 28
  arithmetic, 29, 30
  geometric, 33, 35
  harmonic, 31, 32
  weighted arithmetic, 30
  weighted geometric, 33, 35
  weighted harmonic, 32
memory, 131
  allocation, 144, 245
  bandwidth, 63
  bank, 16
  cycle, 11, 101, 138, 142, 144, 162, 167, 182, 188, 300
  demand from pipelined processor, 237
  desiderata, 131
  DRAM page mode, 169–71
  drum, 16, 25
  EDO DRAM, 170, 171
  hierarchical, 4, 132
  interleaved, 163
    C access, 164, 167
    S access, 167
    wide word, 167, 169
  latency, 135, 137, 143–5, 147, 162, 163, 295
  main, 6, 17, 54, 104, 132, 248, 252
  module, 100, 164, 166, 167
  MOS DRAM, 167
  nibble mode DRAM, 169–71
  protection, 248, 275, 290, 293
  random access, 6, 179
  real, 136
  support, 283
  synchronous link DRAM, 170
  virtual, 13, 50, 92, 101, 133–9, 142, 143, 145, 148–50, 160, 161, 171–4, 177–9, 181, 182, 241, 275, 278, 281, 283, 285, 286, 289, 290, 301, 304, 309
memory to memory architecture, 54, 55
MFLOPS, 46
micro instruction set architecture, 194
micro program counter, 194

Petitioner Exhibit 1053, Page 326 of 334

microinstructions, 199, 202
microprocessors, 2, 49, 51, 78, 145, 147, 185, 239
microprogram, 193, 194, 196–203
microprogram, size, 200
microprogramming, 4, 12, 183, 194, 201
Millions of Instructions Per Second (MIPS), 45, 46, 48, 49, 64, 67, 94, 96, 98, 101, 102, 103, 105, 106, 108, 109, 127, 144, 145, 146, 178, 179, 205, 237, 243, 251
Millions of Integer Operations Per Second (MIOPS), 305, 306, 309
MIPS R2000, 64, 65, 67, 94, 96, 98, 101–3, 105, 106, 108, 109, 205
models, 26, 154, 205, 239, 273
model-specific-registers (MSRs), 307
modems, 5, 47, 241, 259, 269
Moore's law, 39, 40, 42, 169
Moore School, 21, 24, 127, 128
Morse Code, 94
Most Significant Bit (MSB), 7, 75, 80, 86, 94, 110, 118, 166, 167, 287, 288
mouse, 259
MU1, 66
multiplication, 20, 54, 62, 102, 104, 105, 114, 115, 118, 119, 121, 123, 126, 306
multiprocessor, 298, 301
multiprogramming, 144, 177, 247

name translation, 135, 137, 138, 139, 143, 144, 163, 227
no write allocate, 154
Not A Number, 126

offset, 60
op-code, 7, 11, 52, 54, 56, 59, 62, 72, 93–6, 98, 101–3, 128–30, 185, 191, 195–7, 199, 217, 218, 232, 276, 278, 281, 291, 301, 302
operating system, 4, 47, 50, 70, 71, 73, 74, 92, 133, 143, 163, 173, 174, 177, 179, 248, 249, 267, 269, 275–9, 291–3, 297
operating system kernel support, 276
operations, 5–8, 13–16, 18, 19, 21, 22, 30, 35, 46, 50–2, 54–6, 59–65, 73, 77, 80–2, 85, 87, 90, 91, 93–6, 98–100, 102, 104, 108–10, 114–16, 118–23, 126, 129, 130, 132, 135, 150, 161, 168–70, 180, 185, 191, 195–7, 203–5, 225, 226, 237, 242–4, 246–8, 251, 264, 275–9, 283, 285, 290, 293, 294, 297, 299–306, 308
  biased exponents, 119
  MMX, 303, 304
ordinal, 75, 305
out of order, 230, 246
overflow, 85, 91, 110, 115, 278, 305

page fault, 134
  fault rate, 134
  handler, 135, 144, 177, 178, 281
  interrupt, 73
page frame, 133, 136, 137, 139, 140, 142, 144, 145
page name, 135–41, 144, 161, 163, 176
page table, 137–41, 142, 144, 150, 161–3, 175, 176, 180, 182–5, 287–90, 297, 298
  direct, 137, 138, 139, 142
  inverted, 139, 141, 142, 144, 150, 163, 180, 182
parallel memory, 164
parity, 90, 258, 261, 262
Pascal, Blaise, 14, 24
percent difference, 27
performance, Clocks Per Instruction (CPI), 27, 30, 33, 34, 46, 49, 109, 193, 199, 200, 205, 207, 211–18, 223, 224, 226–8, 234, 235, 237, 239, 240, 304
  steady state, 45
  transient, 46, 47, 264
performance improvements, 48
performance models, 26, 205, 239
PicoJava, 309
pipeline, 18, 19, 46, 70, 171, 172, 173, 181, 204, 205, 207, 209, 210, 211, 212, 221, 225, 226, 228, 231, 233, 234, 237, 238, 240, 290, 304, 306
  base, 204, 205, 210, 212, 226, 228, 234, 240
  delays, 211
  model, 210
  optimum number of stages, 207, 209, 239
  partitioning, 207, 210
  underpipelined, 209
Poisson distribution, 266, 274
polling, 51, 71, 73, 74, 25–6
  global counter, 254
pop, 54
present bit, 137, 285
Princeton University, 2, 5, 6, 24
priority, 71, 73, 146, 225
probe, 141, 142, 206
processor control design, 183
program counter, 2, 4, 10, 13, 52, 53, 63–72, 88, 91, 102, 130, 135, 155, 185, 186, 194, 197, 199, 203, 205, 211, 212, 225, 232, 233, 248, 293
Program Status Word (PSW), 88, 91, 276–9, 284, 291, 292, 294, 300

queuing models, 265
  closed system, 265, 271
  duty cycle, 267, 268, 270, 271
  open system, 265
  service rate, 266, 270
  service time, 270, 271
  think time, 271–4
  wait time, 271–3

Petitioner Exhibit 1053, Page 327 of 334

Quiet Not A Number (QNAN), 126
quotient, 8, 102, 114, 116, 136

R&D, 41, 44
range, 7, 62, 74, 79, 81–4, 86, 88, 110, 120, 128, 130, 145, 159, 161, 177, 193, 236, 246, 249
rate, 27, 37, 46, 134
ratio, 29, 33, 35, 146
read, 1, 4, 63, 99, 102, 146, 153, 179, 195, 198, 204, 219, 226, 227, 230, 234, 238, 244, 246, 278
Reduced Instruction Set Computer (RISC), 6, 50, 51, 57, 98, 101, 102, 106, 108, 109, 127, 209, 212, 233
registers, 2, 3, 4, 6, 7, 8, 13, 18, 19, 20, 52, 54–63, 65–7, 69, 71–3, 82, 88, 90–3, 96, 97–102, 106, 109, 123, 130, 132, 138, 144, 146, 168, 174, 176, 183, 184–6, 188, 191, 192, 195–7, 199, 202, 204, 205, 207, 208, 210, 226, 228, 229, 230, 231, 234, 236–40, 242–7, 276–81, 285–7, 291, 292, 294, 298, 302–7, 309
  architected, 66, 73, 102, 184, 242
  control, 277, 278, 281, 285, 287
  debug, 307
  demand from pipelined processors, 238
  flag, 277, 280, 281, 283
  implemented, 8, 13, 82, 184, 276, 306,
  performance monitoring, 307
renaming, 230, 231
request rate, 270
reservation tables, 206, 212, 214, 215, 228, 234, 239
return address, store, 70
rounding, 42, 115, 12–16, 130, 256
RS232 serial protocol, 261

saved program counter, 67, 232, 233
saving processor state, 69
saving state, 71
segment table, 175, 182, 284, 285
segmentation, 174, 176
segmentation, paged, 174
segmentation table, global, 254, 287
select, late, 150, 155
self-modifying programs, 60, 65, 66, 70, 131, 146, 219
semaphores, 299, 302
Sequence Controlled Automatic Calculator, 20
serial communication, 260
serial execution model, 70, 145, 183, 204, 207, 225, 227, 231
server, 46, 47, 249, 250, 266, 269, 270
set up time, 209
side effect, 52, 63, 65, 90–3, 102, 163, 184, 225, 233, 277, 283, 290, 295, 300, 302
signed, 75, 80, 103
significand, 85, 86, 118, 119, 121–6, 130, 303
Silent Not A Number (SNAN), 126
SIMD, 304, 307
simulation, 26, 265
SLDRAM, 171, 181
SPARC, 60, 105, 106, 108, 109, 127, 146
speedup, 27–9, 33–5, 37–9, 44, 48, 49, 105, 107, 144, 159, 160, 166, 167, 179, 202, 207, 216, 227, 236, 270, 273, 274
stack, 54, 56, 57, 62, 63, 66–9, 71, 88, 100, 106, 123, 202, 283, 292, 303
  evaluation, 54, 56, 57, 100, 123
  procedure, 67, 283
state assignment, 186, 190, 192
state diagram, 186, 190
state machine, 3, 4, 12, 120, 184–6, 190, 192, 193, 194, 199, 202, 203, 249, 253
state, process, 3, 4, 53, 66, 69–71, 131
state, processor, 4, 53, 66, 67, 69, 70, 71, 72, 221, 233, 276, 280, 292
state transition, 186, 191, 192, 202
statistics, high level language, 105
statistics, instruction, 103, 106
statistics, instruction dynamic, 104–6, 146
statistics, instruction static, 106
sticky bit, 121–5, 130
subroutine, 13, 14, 18, 50, 60, 62–71, 81, 88, 202, 243, 247, 277
  call, 63, 65–70, 106, 126, 135, 202, 247, 276, 277
  open, 243
  recursive, 65
  reusable, 65
subtract, 8, 52, 106, 111, 116, 119, 120
Sun Microsystems, 60, 108
superscalar, 205, 234, 239
  degree, 234
  instruction alignment, 234
synchronization, 159, 161, 166, 242, 248, 259, 260–4, 298–302, 309
synchronizer, 264, 265
synchronous, 70, 170, 232, 252, 256, 257, 260, 261, 262
synonyms, 181

tables of functions, 14, 19, 21
tag, 2, 4, 137, 141, 149–1, 153, 155, 178, 182, 285
taxonomy, 6, 64, 72, 74, 75, 241, 250, 258
terminals, 47, 250–2, 271–4
Test and Set, 300, 301, 302
Texas Instruments, 99
Texas Instruments ASC, 74, 105
time stamp counter, 307
transaction processing, 250
translate, 135, 177, 283
Translation Lookaside Buffer (TLB), 144, 161–3, 175, 176, 181, 182, 227, 284, 285, 289, 290

Petitioner Exhibit 1053, Page 328 of 334

transport time, 143, 154, 155, 158–61, 164, 165, 177–9, 240
traps, 70

underpipelined, 209
Univac 1103, 246
University of Pennsylvania, 2, 21, 128
untranslated, 148, 171, 285

variable length instruction, 94, 95, 100, 101, 130
vector, 12, 13, 302
virtual memory, 133
  paged, 133
  performance, 142
  support, 283
VLSI, 47, 170
von Neumann architecture, 2, 5
von Neumann, John, 2, 3, 5, 6, 7, 8, 10, 12–4, 20, 24, 50, 54, 60, 65, 70, 77, 90, 128, 131, 203, 243, 274

Wang Laboratories, 1
window, 234
Windows95, 179, 244
Windows98, 241
workload, 26, 215, 238, 240
World Wide Web, 1, 47, 252, 269, 274
write, 4, 24, 25, 99, 102, 146, 147, 153, 154, 179, 180, 186, 192, 196, 198, 202, 205, 227, 229, 230, 231, 234, 238, 244, 246, 274, 278, 309
write allocate, 154
write back, 154, 180, 234
write through, 154, 180, 226

Zuse, Konrad, 2, 14, 17, 19, 24, 77, 81, 86
Zuse, Z1, 17, 24
Zuse, Z2, 17
Zuse, Z3, 2, 17, 20, 23, 24, 25
Zuse, Z4, 81, 86

Petitioner Exhibit 1053, Page 329 of 334

TRADEMARKS

K5 is a trademark of Advanced Micro Devices
CDC6600 is a trademark of Control Data Corporation
LSI-11, PDP-11, and VAX are trademarks of Digital Equipment Corporation
8080, 8086, 286, Pentium, and MMX are trademarks of Intel Corporation
704, S360, S370, S390, and PowerPC are trademarks of International Business Machines
Zip is a trademark of Iomega Corporation
MS Dos, Windows95, and Windows98 are trademarks of Microsoft Corporation
MIPS R2000, and MIPS-T5 are trademarks of MIPS Computer Corporation
MC68000 and MC88110 are trademarks of Motorola, Inc.
NS 16000 is a trademark of National Semiconductor Corporation
SPARC is a trademark of Sun Microsystems
TI 9900 is a trademark of Texas Instruments Corporation
Z8000 is a trademark of Zilog Corporation

Petitioner Exhibit 1053, Page 330 of 334



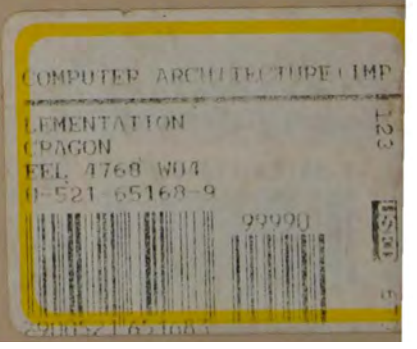


This textbook provides a clear and concise introduction to computer architecture and implementation. Two important themes are interwoven throughout the book. The first is an overview of the major concepts and design philosophies of computer architecture and organization. The second is the early introduction and use of analytic modeling of computer performance.

The author begins by describing the classic von Neumann architecture and then presents in detail a number of performance models and evaluation techniques. He goes on to cover user instruction set design, including RISC architecture. A unique feature of the book is its memory-centric approach – memory systems are discussed before processor implementations. The author also deals with pipelined processors, input/output techniques, queuing modes, and extended instruction set architectures. Each topic is illustrated with reference to actual IBM and Intel architectures.

The book contains many worked examples and over 130 homework exercises. It is an ideal textbook for a one-semester undergraduate course in computer architecture and implementation.

Harvey G. Cragon holds the Ernest J. Cockrell, Jr., Centennial Chair Emeritus in Engineering at the University of Texas at Austin. He is a member of the U.S. National Academy of Engineering, a Life Fellow of the IEEE, and a Fellow of the Association for Computing Machinery. He has over 30 years' industrial experience, including 25 years with Texas Instruments. He has written over 50 technical papers and holds 9 patents in the field of computer engineering.



Petitioner Exhibit 1053, Page 334 of 334