that resources can be used at 100% of their maximum rate without side effects from interference. A later example takes a more realistic view.

Example

Given the following performance and cost information:

a 50-MIPS CPU costing $50,000

an 8-byte-wide memory with a 200-ns cycle time

80 MB/sec I/O bus with room for 20 SCSI buses and controllers

SCSI buses that can transfer 4 MB/sec and support up to 7 disks per bus (these are also called SCSI *strings*)

a $2500 SCSI controller that adds 2 milliseconds (ms) of overhead to perform a disk I/O

an operating system that uses 10,000 CPU instructions for a disk I/O

a choice of a large disk containing 4 GB or a small disk containing 1 GB, each costing $3 per MB

both disks rotate at 3600 RPM, have a 12-ms average seek time, and can transfer 2MB/sec

the storage capacity must be 100 GB, and

the average I/O size is 8 KB

Evaluate the cost per I/O per second (IOPS) of using small or large drives. Assume that every disk I/O requires an average seek and average rotational delay. Use the optimistic assumption that all devices can be used at 100% of capacity and that the workload is evenly divided between all disks.

Answer

I/O performance is limited by the weakest link in the chain, so we evaluate the maximum performance of each link in the I/O chain for each organization to determine the maximum performance of that organization.

Let's start by calculating the maximum number of IOPS for the CPU, main memory, and I/O bus. The CPU I/O performance is determined by the speed of the CPU and the number of instructions to perform a disk I/O:

$$\text{Maximum IOPS for CPU} = \frac{50 \text{ MIPS}}{10000 \text{ instructions per I/O}} = 5000$$

The maximum performance of the memory system is determined by the memory cycle time, the width of the memory, and the size of the I/O transfers:

$$\text{Maximum IOPS for main memory} = \frac{(1/200 \text{ ns})*8}{8 \text{ KB per I/O}} \approx 5000$$

The I/O bus maximum performance is limited by the bus bandwidth and the size of the I/O:

$$\text{Maximum IOPS for the I/O bus} = \frac{80\text{ MB/sec}}{8\text{ KB per I/O}} \approx 10000$$

Thus, no matter which disk is selected, the CPU and main memory limits the maximum performance to no more than 5000 IOPS.

Now its time to look at the performance of the next link in the I/O chain, the SCSI controllers. The time to transfer 8 KB over the SCSI bus is

$$\text{SCSI bus transfer time} = \frac{8\text{ KB}}{4\text{ MB/sec}} = 2\text{ ms}$$

Adding the 2-ms SCSI controller overhead means 4 ms per I/O, making the maximum rate per controller

$$\text{Maximum IOPS per SCSI controller} = \frac{1}{4\text{ ms}} = 250\text{ IOPS}$$

All the organizations will use several controllers, so 250 IOPS is not the limit for the whole system.

The final link in the chain is the disks themselves. The time for an average disk I/O is

$$\text{I/O time} = 12\text{ ms} + \frac{0.5}{3600\text{ RPM}} + \frac{8\text{ KB}}{2\text{ MB/sec}} = 12{+}8.3{+}\ 4\ = 24.3\text{ ms}$$

so the disk performance is

$$\text{Maximum IOPS (using average seeks) per disk} = \frac{1}{24.3\text{ ms}} \approx 41\text{ IOPS}$$

The number of disks in each organization depends on the size of each disk: 100 GB can be either 25 4-GB disks or 100 1-GB disks. The maximum number of I/Os for all the disks is:

$$\begin{aligned}\text{Maximum IOPS for 25 4-GB disks} &= 25 * 41 = 1025\\ \text{Maximum IOPS for 100 1-GB disks} &= 100 * 41 = 4100\end{aligned}$$

Thus, provided there are enough SCSI strings, the disks become the new limit to maximum performance: 1025 IOPS for the 4-GB disks and 4100 for the 1-GB disks.

While we have determined the performance of each link of the I/O chain, we still have to determine how many SCSI buses and controllers to use and how many disks to connect to each controller, as this may further limit maximum performance. The I/O bus is limited to 20 SCSI controllers and the SCSI standard limits disks to 7 per SCSI string. The minimum number of controllers is for the 4-GB disks

$$\text{Minimum number of SCSI strings for 25 4-GB disks} = \frac{25}{7} \text{ or } 4$$

and for 1-GB disks

$$\text{Minimum number of SCSI strings for 100 1-GB disks} = \frac{100}{7} \text{ or } 15$$

We can calculate the maximum IOPS for each configuration:

$$\text{Maximum IOPS for 4 SCSI strings} = 4 * 250 = 1000 \text{ IOPS}$$

$$\text{Maximum IOPS for 15 SCSI strings} = 15 * 250 = 3750 \text{ IOPS}$$

The maximum performance of this number of controllers is slightly lower than the disk I/O throughput, so let's also calculate the number of controllers so they don't become a bottleneck. One way is to find the number of disks they can support per string:

$$\text{Number of disks per SCSI string at full bandwidth} = \frac{250}{41} = 6.1 \text{ or } 6$$

and then calculate the number of strings:

$$\text{Number of SCSI strings for full bandwidth 4-GB disks} = \frac{25}{6} = 4.1 \text{ or } 5$$

$$\text{Number of SCSI strings for full bandwidth 1-GB disks} = \frac{100}{6} = 16.7 \text{ or } 17$$

This establishes the performance of four organizations: 25 4-GB disks with 4 or 5 SCSI strings and 100 1-GB disks with 15 to 17 SCSI strings. The maximum performance of each option is limited by the bottleneck (in boldface):

4-GB disks, 4 strings = Min(5000,5000,10000,1025,**1000**) = 1000 IOPS

4-GB disks, 5 strings = Min(5000,5000,10000,**1025**,1250) = 1025 IOPS

1-GB disks, 15 strings = Min(5000,5000,10000,4100,**3750**) = 3750 IOPS

1-GB disks, 17 strings = Min(5000,5000,10000,**4100**,4250) = 4100 IOPS

We can now calculate the cost for each organization:

4-GB disks, 4 strings = $50,000 + 4*$2,500 + 25 * (4096*$3) = $367,200

4-GB disks, 5 strings = $50,000 + 5*$2,500 + 25 * (4096*$3) = $369,700

1-GB disks, 15 strings = $50,000 + 15*$2,500 + 100 * (1024*$3) = $394,700

1-GB disks, 17 strings = $50,000 + 17*$2,500 + 100 * (1024*$3) = $399,700

Finally, the cost per IOPS for each of the four configurations is $367, $361, $105, and $97, respectively. Calculating maximum number of average I/Os per second assuming 100% utilization of the critical resources, the best cost/performance is the organization with the small disks and the largest number of controllers. The small disks have 3.4 to 3.8 times better cost/performance than the large disks in this example. The only drawback is that the larger number of disks will affect system availability unless some form of redundancy is added (see pages 520–521).

This above example assumed that resources can be used 100%. It is instructive to see what is the bottleneck in each organization.

Example For the organizations in the last example, calculate the percentage of utilization of each resource in the computer system.

Answer Figure 9.31 gives the answer.

Resource	4-GB disks, 4 strings	4-GB disks, 5 strings	1-GB disks, 15 strings	1-GB disks, 17 strings
CPU	20%	21%	75%	82%
Memory	20%	21%	75%	82%
I/O bus	10%	10%	38%	41%
SCSI buses	100%	82%	100%	96%
Disks	98%	100%	91%	100%

FIGURE 9.31 The percentage of utilization of each resource given the four organizations in the previous example. Either the SCSI buses or the disks are the bottleneck.

In reality buses cannot deliver close to 100% of bandwidth without severe increase in latency and reduction in throughput due to contention. A variety of rules of thumb have been evolved to guide I/O designs:

No I/O bus should be utilized more than 75% to 80%;

No disk string should be utilized more than 40%;

No disk arm should be seeking more than 60% of the time.

Example

Recalculate performance in the example above using these rules of thumb, and show the utilization of each component. Are there other organizations that follow these guidelines and improve performance?

Answer

Figure 9.31 shows that the I/O bus is far below the suggested guidelines, so we concentrate on the utilization of seek and SCSI bus. The utilization of seek time per disk is

$$\frac{\text{Time of average seek}}{\text{Time between I/Os}} = \frac{12\text{ ms}}{\frac{1}{41\text{ IOPS}}} = \frac{12}{24} = 50\%$$

which is below the rule of thumb. The biggest impact is on the SCSI bus:

$$\text{Suggested IOPS per SCSI string} = \frac{1}{4\text{ ms}} * 40\% = 100\text{ IOPS.}$$

With this data we can recalculate IOPS for each organization:

4-GB disks, 4 strings = Min(5000,5000,7500,1025,**400**) = 400 IOPS

4-GB disks, 5 strings = Min(5000,5000,7500,1025,**500**) = 500 IOPS

1-GB disks, 15 strings = Min(5000,5000,7500,4100,**1500**) = 1500 IOPS

1-GB disks, 17 strings = Min(5000,5000,7500,4100,**1700**) = 1700 IOPS

Under these assumptions, the small disks have about 3.0 to 4.2 times the performance of the large disks.

Clearly, the string bandwidth is the bottleneck now. The number of disks per string that would not exceed the guideline is

$$\text{Number of disks per SCSI string at full bandwidth} = \frac{100}{41} = 2.4\text{ or }2$$

and the ideal number of strings is

$$\text{Number of SCSI strings for full bandwidth 4-GB disks} = \frac{25}{2} = 12.5\text{ or }13$$

$$\text{Number of SCSI strings for full bandwidth 1-GB disks} = \frac{100}{2} = 50$$

This suggestion is fine for 4-GB disks, but the I/O bus is limited to 20 SCSI controllers and strings so that becomes the limit for 1-GB disks:

4-GB disks, 13 strings = Min(5000,5000,7500,**1025**,1300) = 1025 IOPS

1-GB disks, 20 strings = Min(5000,5000,7500,4100,**2000**) = 2000 IOPS

We can now calculate the cost for each organization:

4-GB disks, 13 strings = $50,000 + 13*$2,500 + 25 * (4096*$3) = $389,700

1-GB disks, 20 strings = $50,000 + 20*$2,500 + 100 * (1024*$3) = $407,200

In this case the small disks cost 5% more yet have about twice the performance of the large disks. The utilization of each resource is shown in Figure 9.32. It shows that following the rule of thumb of 40% string utilization sets the performance limit in all but one case.

Resource	**4-GB disks, 4 strings**	**4-GB disks, 5 strings**	**1-GB disks, 15 strings**	**1-GB disks, 17 strings**	**4-GB disks, 13 strings**	**1-GB disks, 20 strings**
CPU	8%	10%	30%	34%	21%	40%
Memory	8%	10%	30%	34%	21%	40%
I/O bus	5%	7%	20%	23%	14%	27%
SCSI buses	40%	40%	40%	40%	32%	40%
Disks	39%	49%	37%	41%	100%	49%
Seek utilization	19%	24%	18%	20%	49%	24%
IOPS	400	500	1500	1700	1025	2000

FIGURE 9.32 The percentage of utilization of each resource given the six organizations in this example, which tries to limit utilization of key resources to the rules of thumb given above.

9.9 Putting It All Together: The IBM 3990 Storage Subsystem

If computer architects were polled to select the leading company in I/O design, IBM would win hands down. A good deal of IBM's mainframe business is commercial applications, known to be I/O intensive. While there are graphic devices and networks that can be connected to an IBM mainframe, IBM's reputation comes from disk performance. It is on this aspect that we concentrate in this section.

The IBM 360/370 I/O architecture has evolved over a period of 25 years. Initially, the I/O system was general purpose, and no special attention was paid to any particular device. As it became clear that magnetic disks were the chief consumers of I/O, the IBM 360 was tailored to support fast disk I/O. IBM's dominant philosophy is to choose latency over throughput whenever it makes a difference. IBM almost never uses a large buffer outside the CPU; their goal is to set up a clear path from main memory to the I/O device so that when a device is ready, nothing can get in the way. Perhaps IBM followed a corollary to the quote on page 526: you can buy bandwidth, but you need to design for latency. As a secondary philosophy, the CPU is unburdened as much as possible to allow the CPU to continue with computation while others perform the desired I/O activities.

The example for this section is the high-end IBM 3090 CPU and the 3990 Storage Subsystem. The IBM 3090, models 3090/100 to 3090/600, can contain one to six CPUs. This 18.5-ns-clock-cycle machine has a 16-way interleaved memory that can transfer eight bytes every clock cycle on each of two (3090/100) or four (3090/600) buses. Each 3090 processor has a 64-KB, 4-way–set-associative, write-back cache, and the cache supports pipelined access taking two cycles. Each CPU is rated about 30 IBM MIPS (see page 78), giving at most 180 MIPS to the IBM 3090/600. Surveys of IBM mainframe installations suggest a rule of thumb of about 4 GB of disk storage per MIPS of CPU power (see Section 9.12).

It is only fair warning to say that IBM terminology may not be self-evident, although the ideas are not difficult. Remember that this I/O architecture has evolved since 1964. While there may well be ideas that IBM wouldn't include if they were to start anew, they are able to make this scheme work, and make it work well.

The 3990 I/O Subsystem Data-Transfer Hierarchy and Control Hierarchy

The I/O subsystem is divided into two hierarchies:

1. Control—This hierarchy of controllers negotiates a path through a maze of possible connections between the memory and the I/O device and controls the timing of the transfer.

2. Data—This hierarchy of connections is the path over which data flows between memory and the I/O device.

After going over each of the hierarchies, we trace a disk read to help understand the function of each component.

For simplicity, we begin by discussing the data-transfer hierarchy, shown in Figure 9.33 (page 548). This figure shows one section of the hierarchy that contains up to 64 large IBM disks; using 64 of the recently announced IBM 3390 disks, this piece could connect to over one trillion bytes of storage! Yet this piece represents only one-sixth of the capacity of the IBM 3090/600 CPU. This ability to expand from a small I/O system to hundreds of disks and terabytes of storage is what gives IBM mainframes their reputation in the I/O world.

The best-known member of the data hierarchy is the *channel.* The channel is nothing more than 50 wires that connect two levels on the I/O hierarchy together. Only 18 of the 50 wires are used for transferring data (8 data plus 1 parity in each direction), while the rest are for control information. For years the maximum data rate was 3 MB per second, but it recently was raised to 4.5 MB per second. Up to 48 channels can be connected to a 3090/100 CPU, and up to

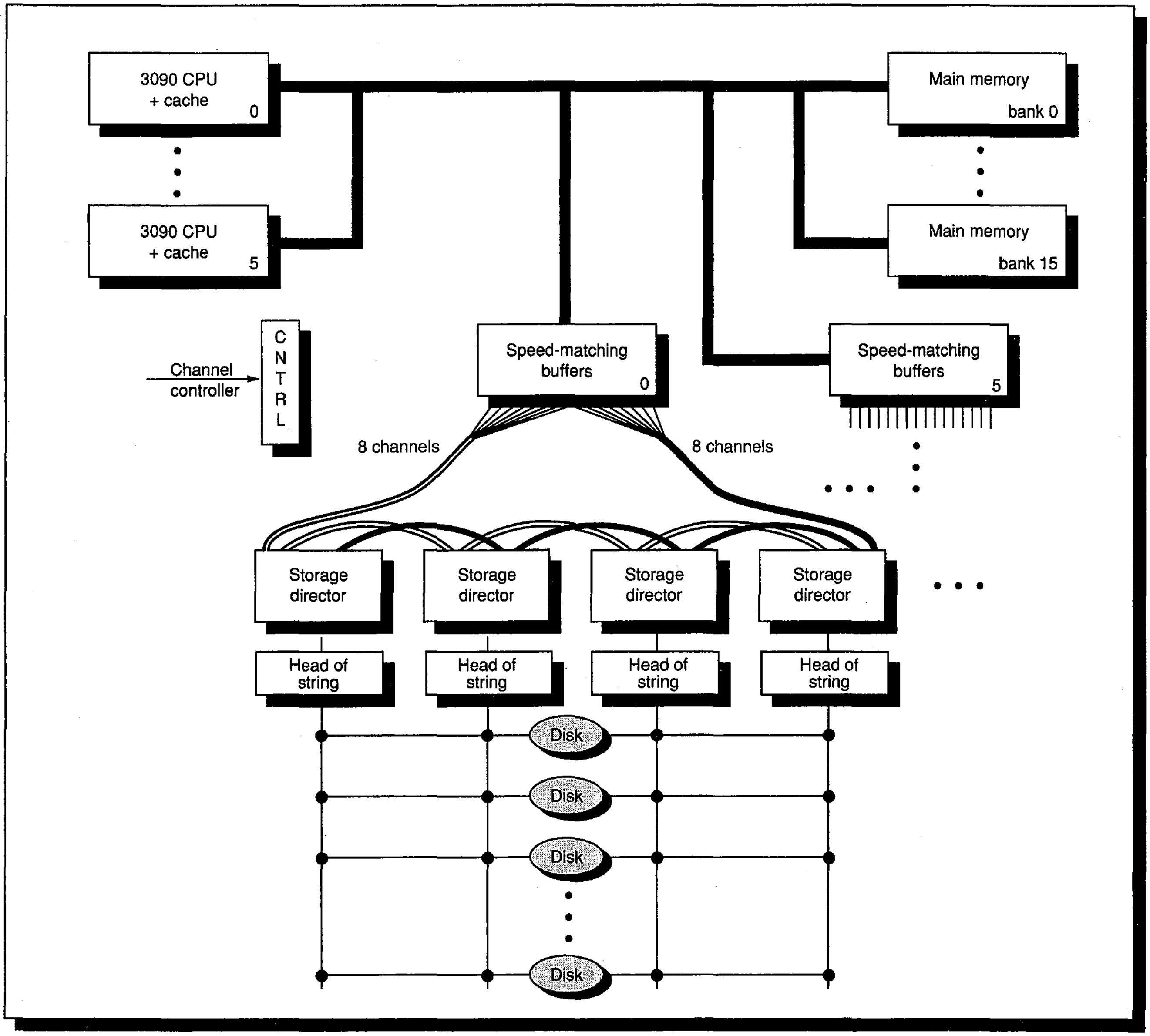


FIGURE 9.33 The data-transfer hierarchy in the IBM 3990 I/O Subsystem. Note that all the channels are connected to all the storage directors. The disks at the bottom represent the quad-ported IBM 3380 disk drives, with the maximum of 64 disks. The collection of disks on the same path to the head-of-string controller is called a *string*.

96 channels to a 3090/600. Because they are "multiprogrammed," channels can actually service several disks. For historical reasons, IBM calls this *block multiplexing*.

Channels are connected to the 3090 main memory via two *speed-matching buffers*, which funnel all the channels into a single port to main memory. Such buffers simply match the bandwidth of the I/O device to the bandwidth of the memory system. There are two 8-byte buffers per channel.

The next level down the data hierarchy is the *storage director*. This is an intermediary device that allows the many channels to talk to many different I/O devices. Four to sixteen channels go to the storage director depending on the model, and two or four paths come out the bottom to the disks. These are called *two-path strings* or *four-path strings* in IBM parlance. Thus, each storage director can talk to any of the disks using one of the strings. At the top of each string is the *head of string*, and all communication between disks and control units must pass through it.

At the bottom of the datapath hierarchy are the disk devices themselves. To increase availability, disk devices like the IBM 3380 provide four paths to connect to the storage director; if one path fails, the device can still be connected.

The redundant paths from main memory to the I/O device not only improve availability, but also can improve performance. Since the IBM philosophy is to avoid large buffers, the path from the I/O device to main memory must remain connected until the transfer is complete. If there were a single hierarchical path from devices to the speed-matching buffer, only one I/O device in a subtree could transfer at a time. Instead, the multiple paths allow multiple devices to transfer simultaneously through the storage director and into memory.

The task of setting up the datapath connection is that of the control hierarchy. Figure 9.34 shows both the control and data hierarchies of the 3990 I/O subsystem. The new device is the I/O processor. The 3090 channel controller and I/O processor are load/store machines similar to DLX, except that there is no memory hierarchy. In the next subsection we see how the two hierarchies work together to read a disk sector.

Tracing a Disk Read in the IBM 3990 I/O Subsystem

The 12 steps below trace a sector read from an IBM 3380 disk. Each of the 12 steps is labeled on a drawing of the full hierarchy in Figure 9.34 (page 550).

1. The user sets up a data structure in memory containing the operations that should occur during this I/O event. This data structure is termed an *I/O control block*, or IOCB, which also points to a list of channel control words (CCWs). This list is called a *channel program*. Normally, the operating system provides the channel program, but some users write their own. The operating system checks the IOCB for protection violations before the I/O can continue.

2. The CPU executes a START SUBCHANNEL instruction. The actual request is defined in the channel program. A channel program to read a record might look like Figure 9.35.

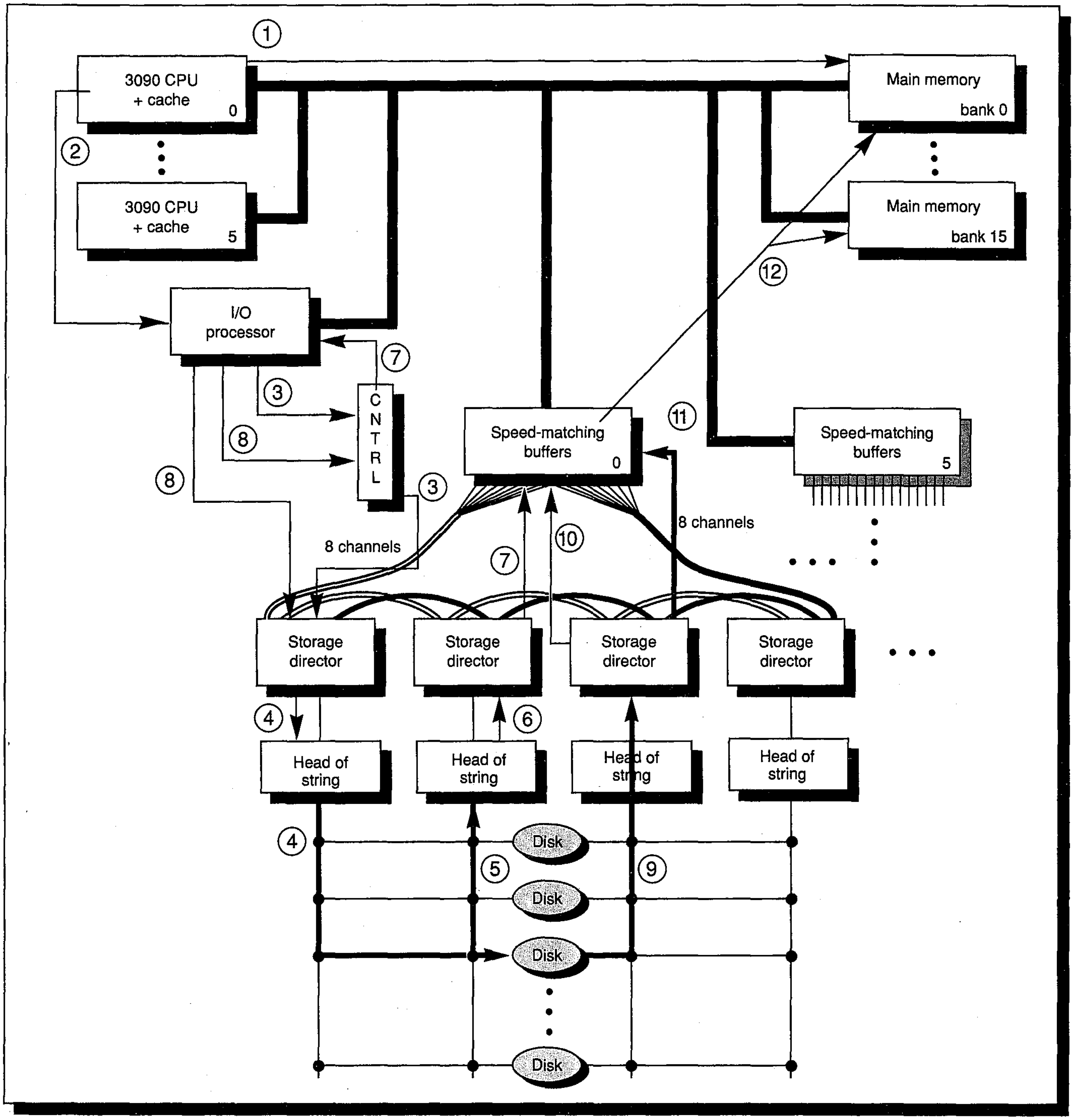


FIGURE 9.34 The control and data hierarchies in the IBM 3990 I/O Subsystem labeled with the 12 steps to read a sector from disk. The only new box over Figure 9.33 (page 548) is the I/O processor.

Location	CCW	Comment
CCW1:	Define Extent	Transfers a 16-byte parameter to the storage director. The channel sees this as a write data transfer.
CCW2:	Locate Record	Transfers a 16-byte parameter to the storage director as above. The parameter identifies the operation (read in this case) plus seek, sector number, and record ID. The channel again sees this as a write data transfer.
CCW3:	Read Data	Transfers the desired disk data to the channel and then to the main memory.

FIGURE 9.35 A channel program to perform a disk read, consisting of three channel command words (CCWs). The operating system checks for virtual memory access violations of CCWs by simulating them to check for violations. These instructions are linked so that only one `START SUBCHANNEL` instruction is needed.

3. The I/O processor uses the control wires of one of the channels to tell the storage director which disk is to be accessed and the disk address to be read. The channel is then released.

4. The storage director sends a SEEK command to the head-of-string controller and the head-of-string controller connects to the desired disk, telling it to seek to the appropriate track, and then disconnects. The disconnect occurs between CCW2 and CCW3 in Figure 9.35.

Upon completion of these first four steps of the read, the arm on the disk seeks the correct track on the correct IBM 3380 disk drive. Other I/O operations can use the control and data hierarchy while this disk is seeking and the data is rotating under the read head. The I/O processor thus acts like a multiprogrammed system, working on other requests while waiting for an I/O event to complete.

An interesting question arises: When there are multiple uses for a single disk, what prevents another seek from screwing up the works before the original request can continue with the I/O event in progress? The answer is the disk appears busy to the programs in the 3090 between the time a `START SUBCHANNEL` instruction starts a channel program (step 2) and the end of that channel program. An attempt to execute another `START SUBCHANNEL` instruction would receive busy status from the channel or from the disk device.

After both the seek completes and the disk rotates to the desired point relative to the read head, the disk reconnects to a channel. To determine the rotational position of the 3380 disk, IBM provides rotational positional sensing (RPS), a feature that gives early warning when the data will rotate under the read head. IBM essentially extends the seek time to include some of the rotation time, thereby tying up the datapath as little as possible. Then the I/O can continue:

5. When the disk completes the seek and rotates to the correct position, it contacts the head-of-string controller.

6. The head-of-string controller looks for a free storage director to send the signal that the disk is on the right track.

7. The storage director looks for a free channel so that it can use the control wires to tell the I/O processor that the disk is on the right track.

8. The I/O processor simultaneously contacts the storage director and I/O device (the IBM 3380 disk) to give the OK to transfer data, and tells the channel controller where to put the information in main memory when it arrives at the channel.

There is now a direct path between the I/O device and memory and the transfer can begin:

9. When the disk is ready to transfer, it sends the data at 3 megabytes per second over a bit-serial line to the storage director.

10. The storage director collects 16 bytes in one of two buffers and sends the information on to the channel controller.

11. The channel controller has a pair of 16-byte buffers per storage director and sends 16 bytes over a 3-MB or 4.5-MB per second, 8-bit-wide datapath to the speed-matching buffers.

12. The speed-matching buffers take the information coming in from all channels. There are two 8-byte buffers per channel that send 8 bytes at a time to the appropriate locations in main memory.

Since nothing is free in computer design, one might expect there to be a cost in anticipating the rotational delay using RPS. Sometimes a free path cannot be established in the time available due to other I/O activity, resulting in an *RPS miss*. An RPS miss means the 3990 I/O Subsystem must either:

- Wait another full rotation—16.7 ms—before the data is back under the head, or
- Break down the hierarchical datapath and start all over again!

Lots of RPS misses can ruin response times.

As mentioned above, the IBM I/O system evolved over many years, and Figure 9.36 shows the change in response time for a few of those changes. The first improvement concerns the path for data after reconnection. Before the System/370-XA, the data path through the channels and storage director (steps 5 through 12) had to be the same as the path taken to request the seek (steps 1 through 4). The 370-XA allows the path after reconnection to be different, and this option is called *dynamic path reconnection* (DPR). This change reduced the time waiting for the channel path and the time waiting for disks (queueing delay), yielding a reduction in the total average response time of 17%. The second change in Figure 9.36 involved a new disk design. Improvements to the microcode control of the 3380D made slight improvements in seek time plus removed a restriction that disk arms that were on the same internal path were prevented from operating at the same time. IBM calls this option *Device Level Select* (DLS). This change reduced internal path delays to 0. This had little impact since there was not much time waiting on internal delays because customers intentionally placed data on disks trying to avoid internal path delays. This second change reduced response time another 9%. The final change was addition of a 32-MB write-through disk cache to a 3380D, called the IBM 3880-23. The disk cache reduced average rotational latency, seek time, and queueing delays, giving another 41% reduction in response time.

One indication of the effectiveness of DPR is the number of disk devices connected to a string. Studies of IBM systems using DPR, which average 16 disk devices per string versus 12 without DPR, suggest dynamic reconnect allows a higher I/O rate with comparable response time [Henly and McNutt 1989].

Summary of the IBM 3990 I/O Subsystem

Goals for I/O systems consist of supporting the following:

- Low cost
- A variety of types of I/O devices

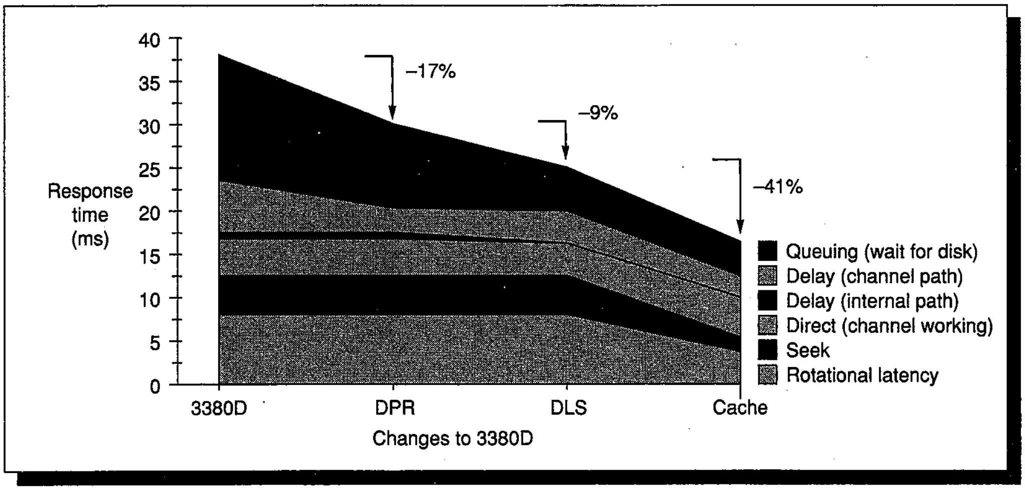


FIGURE 9.36 Changes in response time with improvements in 3380D broken into six categories [Friesenborg and Wicks 1985]. Queueing delay refers to the time when the program waits for another program to finish with the disk. Channel-path delay is the time the operation waits due to the channel path and storage director being busy with another task. Internal-path delay is similar to channel-path delay except it refers to internal paths in the 3380D. Direct means the time the channel path is busy with the operation. Seek time and rotational latency are the standard definitions. Robinson and Blount [1986] report in the study of the 3880-23 that the read hit rate for the 32-MB write-through cache in some large systems averages about 90%, with reads accounting for 92% of the disk accesses.

- A large number of I/O devices at a time
- High performance
- Low latency

Substantial expendability and lower latency are hard to get at the same time. IBM channel-based systems achieve the third and fourth goals by utilizing hierarchical data paths to connect a large number of devices. The many devices and parallel paths allow simultaneous transfers and, thus, high throughput. By avoiding large buffers and providing enough extra paths to minimize delay from congestion, channels offer low-latency I/O as well. To maximize use of the hierarchy, IBM uses rotational positional sensing to extend the time that other tasks can use the hierarchy during an I/O operation.

Therefore, a key to performance of the IBM I/O subsystem is the number of rotational positional misses and congestion on the channel paths. A rule of thumb is that the single-path channels should be no more than 30% utilized and the quad-path channels should be no more than 60% utilized, or too many rotational positional misses will result. This I/O architecture dominates the industry, yet it would be interesting to see what, if anything, IBM would do differently if given a clean slate.

9.10 Fallacies and Pitfalls

Fallacy: I/O plays a small role in supercomputer design

The goal of the Illiac IV was to be the world's fastest computer. It may not have achieved that goal, but it showed I/O as the Achilles' Heel of high-performance machines. In some tasks, more time was spent in loading data than in computing. Amdahl's Law demonstrated the importance of high performance in all the parts of a high-speed computer. (In fact, Amdahl made his comment in reaction to claims for performance through parallelism made on behalf of the Illiac IV.) The Illiac IV had a very fast transfer rate (60 MB/sec), but very small, fixed-head disks (12-MB capacity). Since they were not large enough, more storage was provided on a separate computer. This led to two ways of measuring I/O overhead:

> *Warm start*—Assuming the data is on the fast, small disks, I/O overhead is the time to load the Illiac IV memory from those disks.
>
> *Cold start*—Assuming the data is in on the other computer, I/O overhead must include the time to first transfer the data to the Illiac IV fast disks.

Figure 9.37 shows ten applications written for the Illiac IV in 1979. Assuming warm starts, the supercomputer was busy 78% of the time and waiting for I/O 22% of the time; assuming cold starts, it was busy 59% of the time and waiting for I/O 41% of the time.

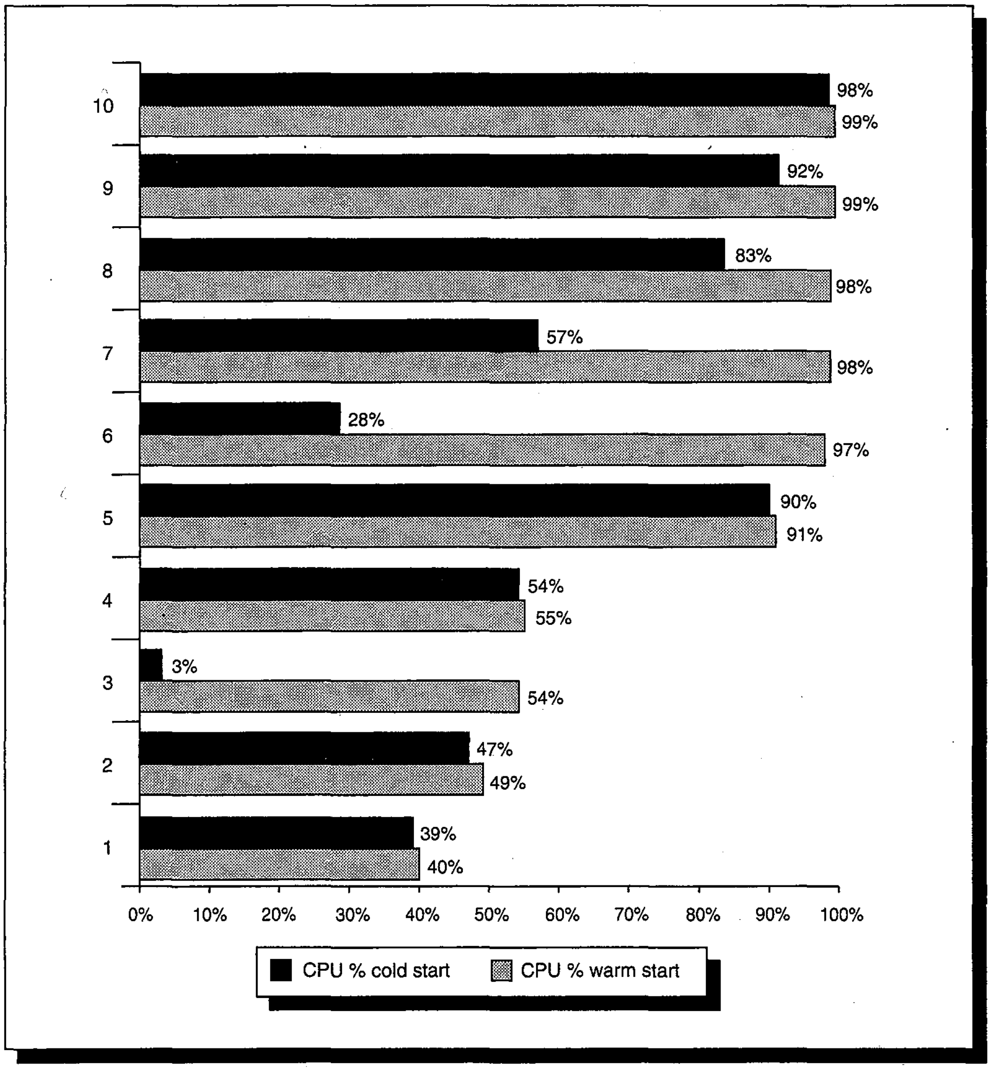


FIGURE 9.37 Feierback and Stevenson [1979] summarized the important Illiac IV applications and the percentage of time spent computing versus waiting for I/O. The arithmetic means of the 10 programs are 78% computing for warm start and 59% computing for cold start.

Pitfall: Moving functions from the CPU to the I/O processor to improve performance.

There are many examples of this pitfall, although I/O processors can enhance performance. A problem inherent with a family of computers is that the migration of an I/O feature usually changes the instruction set architecture or system architecture in a programmer-visible way, causing all future machines to have to live with a decision that made sense in the past. If CPUs are improved in cost/performance more rapidly than the I/O processor (and this will likely be the case) then moving the function may result in a slower machine in the next CPU.

The most telling example comes from the IBM 360. It was decided that the performance of the ISAM system, an early database system, would improve if some of the record searching occurred in the disk controller itself. A key field was associated with each record, and the device searched each key as the disk rotated until it found a match. It would then transfer the desired record. For the disk to find the key, there had to be an extra gap in the track. This scheme is applicable to searches through indices as well as data.

The speed a track can be searched is limited by the speed of the disk and of the number of keys that can be packed on a track. On an IBM 3330 disk the key is typically 10 characters, but the total gap between records is equivalent to 191 characters if there were a key. (The gap is only 135 characters if there is no key, since there is no need for an extra gap for the key.) If we assume the data is also 10 characters and the track has nothing else on it, then a 13165-byte track can contain

$$\frac{13165}{191+10+10} = 62 \text{ key-data records}$$

This performance is

$$\frac{16.7 \text{ ms (1 revolution)}}{62} \approx .25 \text{ ms/key search}$$

In place of this scheme, we could put several key-data pairs in a single block and have smaller inter-record gaps. Assuming there are 15 key-data pairs per block and the track has nothing else on it, then

$$\frac{13165}{135+15*(10+10)} = \frac{13165}{135+300} = 30 \textbf{ blocks} \text{ of key-data pairs}$$

The revised performance is then

$$\frac{16.7 \text{ ms (1 revolution)}}{30*15} \approx .04 \text{ ms/key search}$$

Yet as CPUs got faster, the CPU time for a search was trivial. While the strategy made early machines faster, programs that use the search-key operation in the I/O processor run six times slower on today's machines!

Fallacy: Comparing the price of media versus the price of the packaged system.

This happens most frequently when new memory technologies are compared to magnetic disks. For example, comparing the DRAM-chip price to magnetic-disk packaged price in Figure 9.16 (page 518) suggests the difference is less than a factor of 10, but its much greater when the price of packaging DRAM is included. A common mistake with removable media is to compare the media cost not including the drive to read the media. For example, optical media costs only $1 per MB in 1990, but including the cost of the optical drive may bring the price closer to $6 per MB.

Fallacy: The time of an average seek of a disk in a computer system is the time for a seek of one-third the number of cylinders.

This fallacy comes from confusing the way manufacturers market disks with the expected performance and with the false assumption that seek times are linear in distance. The 1/3 distance rule of thumb comes from calculating the distance of a seek from one random location to another random location, not including the current cylinder and assuming there are a large number of cylinders. In the past, manufacturers listed the seek of this distance to offer a consistent basis for comparison. (As mentioned on page 516, today they calculate the "average" by timing all seeks and dividing by the number.) Assuming (incorrectly) that seek time is linear in distance, and using the manufacturers reported minimum and "average" seek times, a common technique to predict seek time is:

$$\text{Time}_{\text{seek}} = \text{Time}_{\text{minimum}} + \frac{\text{Distance}}{\text{Distance}_{\text{average}}} * (\text{Time}_{\text{average}} - \text{Time}_{\text{minimum}})$$

The fallacy concerning seek time is twofold. First, seek time is **not** linear with distance; the arm must accelerate to overcome inertia, reach its maximum traveling speed, decelerate as it reaches the requested position, and then wait to allow the arm to stop vibrating (settle time). Moreover, in recent disks sometimes the arm must pause to control vibrations. Figure 9.38 (page 558) plots time versus seek distance for an example disk. It also shows the error in the simple seek-time formula above. For short seeks, the acceleration phase plays a larger role than the maximum traveling speed, and this phase is typically modeled as the square root of the distance. Figure 9.39 (page 558) shows accurate formulas used to model the seek time versus distance for two disks.

The second problem is the average in the product specification would only be true if there was no locality to disk activity. Fortunately, there is both temporal and spatial locality (page 403 in Chapter 8): disk blocks get used more than once and disk blocks near the current cylinder are more likely to be used than those farther away. For example, Figure 9.40 (page 559) shows sample measurements of seek distances for two workloads: a UNIX timesharing workload and a business-processing workload. Notice the high percentage of disk accesses to the same cylinder, labeled distance 0 in the graphs, in both workloads.

Thus, this fallacy couldn't be more misleading. The Exercises debunk this fallacy in more detail.

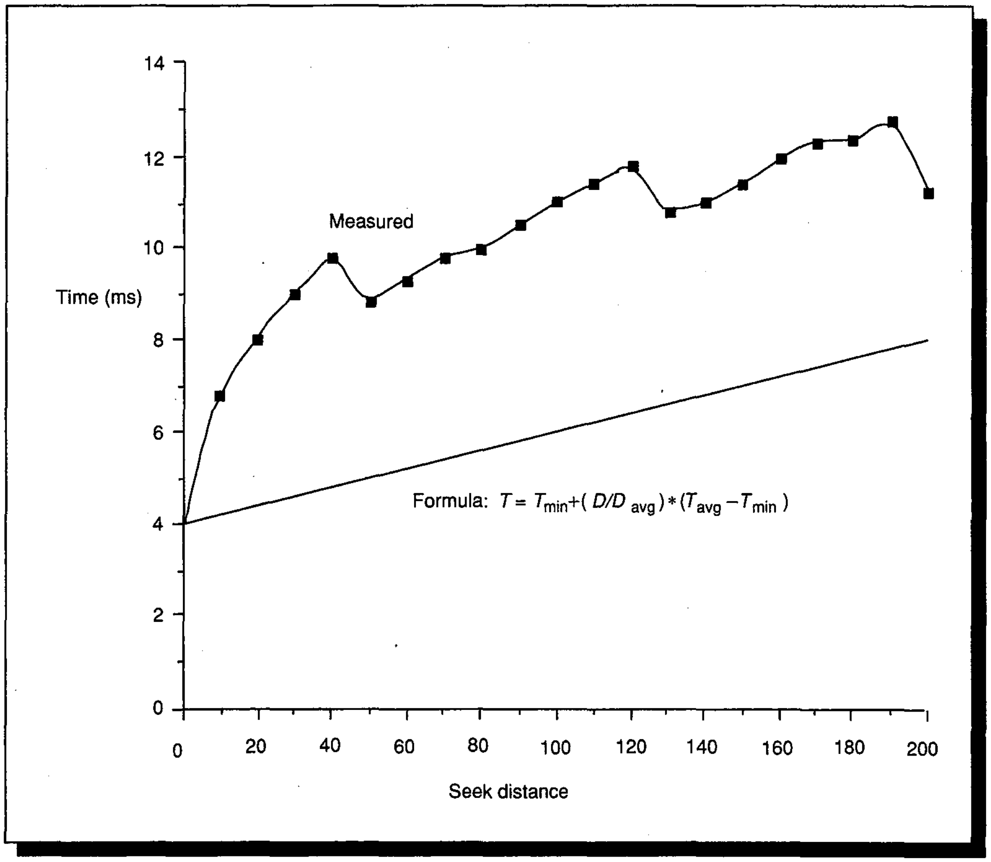


FIGURE 9.38 Seek time versus seek distance for the first 200 cylinders. The Imprimis Sabre 97209 contains 1.2 GB using 1635 cylinders and has the IPI-2 interface [Imprimis 1989]. This is an 8-inch disk. Note that longer seeks can take **less** time than shorter seeks. For example, a 40-cylinder seek takes almost 10 ms, while a 50-cylinder seek takes less than 9 ms.

IBM 3380D Range for formula		Formulas	IBM 3380J Range for formula		Formulas
≥	≤		≥	≤	
1	50	$1.9 + \sqrt{\text{Distance}} - \frac{\text{Distance}}{50}$	1	50	$2.48 + \sqrt{\text{Distance}} - \frac{\text{Distance}}{20}$
51	100	8.1 + 0.044 * (Distance–50)	51	130	7.28 + 0.0320 * (Distance–50)
101	500	10.3 + 0.025 * (Distance–100)	131	500	10.08 + 0.0166 * (Distance–130)
501	884	20.4 + 0.017 * (Distance–500)	501	884	16.00 + 0.0114 * (Distance–500)

FIGURE 9.39 Formulas for seek time in ms for two IBM disks. Thisquen [1988] measured these disks and proposed these formulas to model them. The two columns on the left show the range of seek distances in cylinders to which each formula applies. Each disk has 885 cylinders, so the maximum seek is 884.

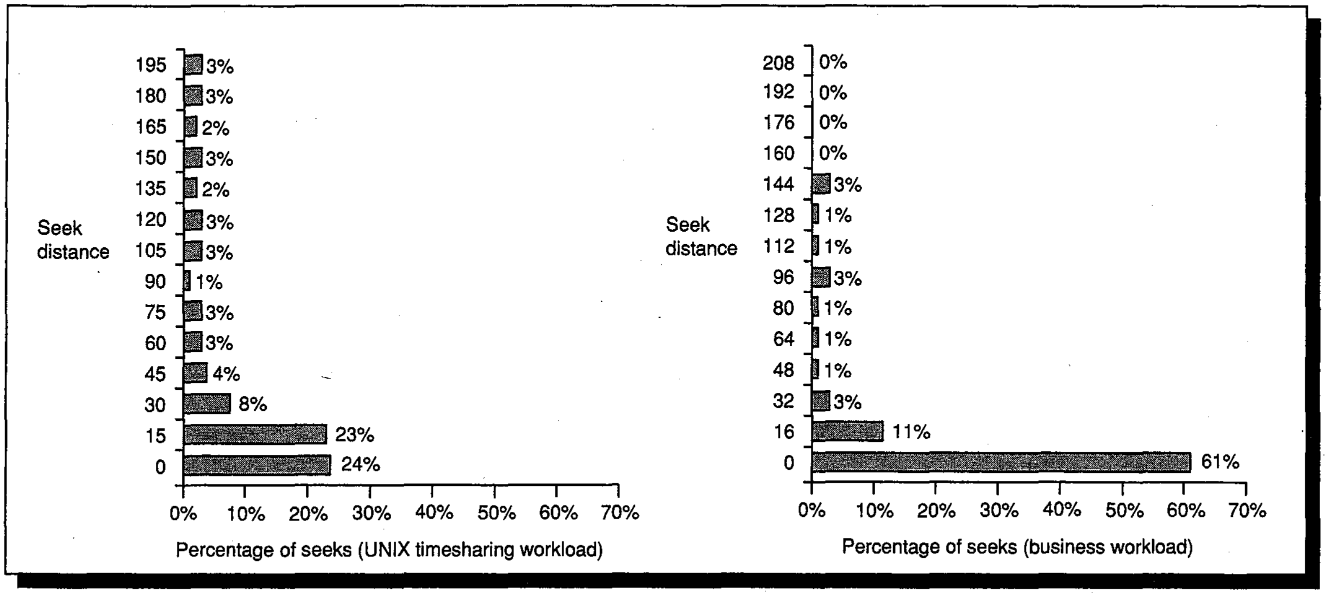


FIGURE 9.40 Sample measurements of seek distances for two systems. The left measurements were taken on a UNIX timesharing system. The right measurements were taken from a business processing application in which the disk seek activity was scheduled. Seek distance of 0 means the access was made to the same cylinder. The rest of the numbers show the collective percentage for distances up between numbers on the y axis. For example, 11% for the bar labeled 16 in the business graph means that the percentage of seeks between 1 and 16 cylinders was 11%. The UNIX measurements stopped at 200 cylinders, but this captured 85% of the accesses. The total was 1000 cylinders. The business measurements tracked all 816 cylinders of the disks. The only seek distances with 1% or greater of the seeks that are not in the graph are 224 with 4% and 304, 336, 512, and 624 each having 1%. This total is 94%, with the difference being small but nonzero distances in other categories. The measurements are courtesy of Dave Anderson of Imprimis.

9.11 Concluding Remarks

I/O systems are judged by the variety of I/O devices, the maximum number of I/O devices, cost, and performance, measured both in latency and in throughput. These common goals lead to widely varying schemes, with some relying extensively on buffering and some avoiding buffering at all costs. If one is clearly better than the other, it is not obvious today. Perhaps this situation is like the instruction set debates of the 1980s, and the strengths and weaknesses of the alternatives will become apparent in the 1990s.

According to Amdahl's Law, ignorance of I/O will lead to wasted performance as CPUs get faster. Disk performance is growing at 4% to 6% per year, while CPUs are growing at a much faster rate. The future demands for I/O include better algorithms, better organizations, and more caching in a struggle to keep pace.

9.12 Historical Perspective and References

The forerunner of today's workstations was the Alto developed at Xerox Palo Alto Research Center in 1974 [Thacker et al. 1982]. This machine reversed traditional wisdom, making instruction set interpretation take back seat to the display: the display used half the memory bandwidth of the Alto. In addition to the bit-mapped display, this historic machine had the first Ethernet [Metcalfe and Boggs 1976] and the first laser printer. It also had a mouse, invented earlier by Doug Engelbart of SRI, and a removable cartridge disk. The 16-bit CPU implemented an instruction set similar to the Data General Nova and offered writable control store (see Chapter 5, Section 5.8). In fact, a single microprogrammable engine drove the graphics display, mouse, disks, network, and, when there was nothing else to do, interpreted the instruction set.

The attraction of a personal computer is that you don't have to share it with anyone. This means response time is predictable, unlike timesharing systems. Early experiments in the importance of fast response time were performed by Doherty and Kelisky [1979]. They showed that if computer-system response time increased a second that user think time did also. Thadhani [1981] showed a jump in productivity as computer response times dropped to a second and another jump as they dropped to a half-second. His results inspired a flock of studies, and they supported his observations [IBM 1982]. In fact, some studies were started to disprove his results! Brady [1986] proposed differentiating entry time from think time (since entry time was becoming significant when the two were lumped together) and provided a cognitive model to explain the more than linear relationship between computer response time and user think time.

The ubiquitous microprocessor has inspired not only personal computers in the 1970s, but the current trend to moving controller functions into I/O devices in the late 1980s and 1990s. For example, microcoded routines in a central CPU made sense for the Alto in 1975, but technological changes soon made separate microprogrammable controller I/O devices economical. These were then replaced by the application-specific integrated circuits. I/O devices continued this trend by moving controllers into the devices themselves. These are called *intelligent devices*, and some bus standards (e.g., IPI and SCSI) have been created just for these devices. Intelligent devices can relax the timing constraints by handling many of the low-level tasks and queuing the results. For example, many SCSI-compatible disk drives include a track buffer on the disk itself, supporting read ahead and connect/disconnect. Thus, on a SCSI string some disks can be seeking and others loading their track buffer while one is transferring data from its buffer over the SCSI bus.

Speaking of buses, the first multivendor bus may have been the PDP-11 Unibus in 1970. DEC encouraged other companies to build devices that would plug into their bus, and many companies did. A more recent example is SCSI, which stands for *small computer systems interface*. This bus, originally called SASI, was invented by Shugart and was later standardized by the IEEE. Sometimes buses are developed in academia; the *NuBus* was developed by Steve Ward and his colleagues at MIT and used by several companies. Alas, this open-door policy on buses is in contrast to companies with proprietary buses using patented interfaces, thereby preventing competition from plug-compatible vendors. This practice also raises costs and lowers availability of I/O devices that plug into proprietary buses, since such devices must have an interface designed just for that bus. Levy [1978] has a nice survey on issues in buses.

We must also give a few references to specific I/O devices. Readers interested in the ARPANET should see Kahn [1972]. As mentioned in one of the section quotes, the father of computer graphics is Ivan Sutherland, who received the ACM Turing Award in 1988. Sutherland's Sketchpad system [1963] set the standard for today's interfaces and displays. See Foley and Van Dam [1982] and Newman and Sproull [1979] for more on computer graphics. Scranton, Thompson, and Hunter [1983] were among the first to report the myths concerning seek times and distances for magnetic disks.

Comments on the future of disks can be found in several sources. Goldstein [1987] projects the capacity and I/O rates for IBM mainframe installations in 1995, suggesting that the ratio is no less than 3.7 GB per IBM mainframe MIPS today, and that will grow to 4.5 GB per MIPS in 1995. Frank [1987] speculated on the physical recording density, proposing the MAD formula on disk growth that we used in Section 9.4. Katz, Patterson, and Gibson [1990] survey current high-performance disks and I/O systems and speculate about future systems. The possibility of achieving higher-performance I/O systems using collections of disks is found in papers by Kim [1986], Salem and Garcia-Molina [1986], and Patterson, Gibson, and Katz [1987].

Looking backward rather than forward, the first machine to extend interrupts from detecting arithmetic abnormalities to detecting asynchronous I/O events is credited as the NBS DYSEAC in 1954 [Leiner and Alexander 1954]. The following year the first machine with DMA was operational, the IBM SAGE. Just as today's DMA, the SAGE had address counters that performed block transfers in parallel with CPU operations. The first I/O channel may have been on the IBM 709 in 1957 [Bashe et al. 1981 and 1986]. Smotherman [1989] explores the history of I/O in more depth.

References

ANON ET AL. [1985]. "A measure of transaction processing power," Tandem Tech. Rep. TR 85.2. Also appeared in *Datamation*, April 1, 1985.

BASHE, C. J., W. BUCHHOLZ, G .V. HAWKINS, J .L. INGRAM, AND N. ROCHESTER [1981]. "The architecture of IBM's early computers," *IBM J. of Research and Development* 25:5 (September) 363–375.

BASHE, C. J., L. R. JOHNSON, J. H. PALMER, AND E. W. PUGH [1986]. *IBM's Early Computers*, MIT Press, Cambridge, Mass.

BORRILL, P. L. [1986]. "32-bit buses–An objective comparison," *Proc. Buscon 1986 West*, San Jose, Calif., 138–145.

BRADY, J. T. [1986]. "A theory of productivity in the creative process," *IEEE CG&A* (May) 25–34.

BUCHER, I. V. AND A. H. HAYES [1980]. "I/O Performance measurement on Cray-1 and CDC 7000 computers," *Proc. Computer Performance Evaluation Users Group, 16th Meeting*, NBS 500-65, 245–254.

CHEN, P. [1989]. *An Evaluation of Redundant Arrays of Inexpensive Disks Using an Amdahl 5890*, M. S. Thesis, Computer Science Division, Tech. Rep. UCB/CSD 89/506.

DOHERTY, W. J. AND R. P. KELISKY [1979]. "Managing VM/CMS systems for user effectiveness," *IBM Systems J.* 18:1, 143–166.

FEIERBACK, G AND D. STEVENSON [1979]. "The Illiac-IV," in *Infotech State of the Art Report on Supercomptuers*, Maidenhead, England. This data also appears in D. P. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples* (1982), McGraw-Hill, New York, 268–269.

FOLEY, J. D. AND A. VAN DAM [1982]. *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, Reading, Mass.

FRANK, P. D. [1987]. "Advances in Head Technology," presentation at *Challenges in Winchester Technology* (December 15), Santa Clara Univ.

FRIESENBORG, S. E. AND R. J. WICKS [1985]. "DASD expectations: The 3380, 3380-23, and MVS/XA," Tech. Bulletin GG22-9363-02 (July 10), Washington Systems Center.

GOLDSTEIN, S. [1987]. "Storage performance—an eight year outlook," Tech. Rep. TR 03.308-1 (October), Santa Teresa Laboratory, IBM, San Jose, Calif.

HENLY, M. AND B. MCNUTT [1989]. "DASD I/O characteristics: A comparison of MVS to VM," Tech. Rep. TR 02.1550 (May), IBM, General Products Division, San Jose, Calif.

HOWARD, J. H. ET AL. [1988]. "Scale and performance in a distributed file system," *ACM Trans. on Computer Systems* 6:1, 51–81.

IBM [1982]. *The Economic Value of Rapid Response Time*, GE20-0752-0 White Plains, N.Y., 11–82.

IMPRIMIS [1989]. "Imprimis Product Specification, 97209 Sabre Disk Drive IPI-2 Interface 1.2 GB," Document No. 64402302 (May).

KAHN, R. E. [1972]. "Resource-sharing computer communication networks," *Proc. IEEE* 60:11 (November) 1397-1407.

KATZ, R. H., D. A. PATTERSON, AND G. A. GIBSON [1990]. "Disk system architectures for high performance computing," *Proc. IEEE* 78:2 (February).

KIM, M. Y. [1986]. "Synchronized disk interleaving," *IEEE Trans. on Computers* C-35:11 (November).

LEINER, A. L. [1954]. "System specifications for the DYSEAC," *J. ACM* 1:2 (April) 57–81.

LEINER, A. L. AND S. N. ALEXANDER [1954]. "System organization of the DYSEAC," *IRE Trans. of Electronic Computers* EC-3:1 (March) 1–10.

LEVY, J. V. [1978]. "Buses: The skeleton of computer structures," in *Computer Engineering: A DEC View of Hardware Systems Design*, C. G. Bell, J. C. Mudge, and J. E. McNamara, eds., Digital Press, Bedford, Mass.

MABERLY, N. C. [1966]. *Mastering Speed Reading*, New American Library, Inc., New York.

METCALFE, R. M. AND D. R. BOGGS [1976]. "Ethernet: Distributed packet switching for local computer networks," *Comm. ACM* 19:7 (July) 395–404.

NEWMAN, W. N. AND R. F. SPROULL [1979]. *Principles of Interactive Computer Graphics*, 2nd ed., McGraw-Hill, New York.

OUSTERHOUT, J. K. ET AL. [1985]. "A trace-driven analysis of the UNIX 4.2 BSD file system," *Proc. Tenth ACM Symposium on Operating Systems Principles*, Orcas Island, Wash., 15–24.

PATTERSON, D. A., G. A. GIBSON, AND R. H. KATZ [1987]. "A case for redundant arrays of inexpensive disks (RAID)," Tech. Rep. UCB/CSD 87/391, Univ. of Calif. Also appeared in *ACM SIGMOD Conf. Proc.*, Chicago, Illinois, June 1–3, 1988, 109–116.

ROBINSON, B. AND L. BLOUNT [1986]. "The VM/HPO 3880-23 performance results," IBM Tech. Bulletin, GG66-0247-00 (April), Washington Systems Center, Gathersburg, Md.

SALEM, K. AND H. GARCIA-MOLINA [1986]. "Disk striping," *IEEE 1986 Int'l Conf. on Data Engineering*.

SCRANTON, R. A., D. A. THOMPSON, AND D. W. HUNTER [1983]. "The access time myth," Tech. Rep. RC 10197 (45223) (September 21), IBM, Yorktown Heights, N.Y.

SMITH, A. J. [1985]. "Disk cache—miss ratio analysis and design considerations," *ACM Trans. on Computer Systems* 3:3 (August) 161–203.

SMOTHERMAN , M. [1989]. "A sequencing-based taxonomy of I/O systems and review of historical machines," *Computer Architecture News* 17:5 (September) 5–15.

SUTHERLAND, I. E. [1963]. "Sketchpad: A man-machine graphical communication system," *Spring Joint Computer Conf.* 329.

THACKER, C. P., E. M. MCCREIGHT, B. W. LAMPSON, R. F. SPROULL, AND D. R. BOGGS [1982]. "Alto: A personal computer," in *Computer Structures: Principles and Examples*, D. P. Siewiorek, C. G. Bell, and A. Newell, eds., McGraw-Hill, New York, 549–572.

THADHANI, A. J. [1981]. "Interactive user productivity," *IBM Systems J.* 20:4, 407–423.

THISQUEN, J. [1988]. "Seek time measurements," *Amdahl Peripheral Products Division Tech. Rep.* (May).

EXERCISES

9.1 <9.10> [10/25/10] Using the formulas in Figure 9.39 (page 558):

a. [10] Calculate the seek time for moving the arm one-third of the cylinders for both disks.

b. [25] Write a program to calculate the "average" seek time by estimating the time for all possible seeks using these formulas and then dividing by the number of seeks.

c. [10] How close does (a) approximate (b)?

9.2 <9.10> [15/20] Using the formulas in Figure 9.39 (page 558) and the statistics in Figure 9.40 (page 559), calculate the average seek distance and the average seek time on the IBM 3380J. Use the midpoint of a range as the seek distance. For example, use 98 as the seek distance for the entry representing 91–105 in Figure 9.40. For the business workload, just ignore the missing 5% of the seeks. For the UNIX workload, assume the missing 15% of the seeks have an average distance of 300 cylinders.

a. [15] If you were misled by the fallacy, you might calculate the average distance as 884/3. What is the measured distance for each workload?

b. [20] The time to seek 884/3 cylinders on the IBM 3380J is about 12.8 ms. What is the average seek time for each workload on the IBM 3380J using the measurements?

9.3 <1.4,8.4,9.4> [20/10/Discussion] Assume the improvements in density of DRAMs and magnetic disks continue as predicted in Figure 1.5 (page 17). Assuming that the improvement in cost per megabyte tracks the density improvements and that 1990 is the start of the 4-megabit DRAM generation, when will the cost per megabyte of DRAM equal the cost per megabyte of magnetic disk given:

- The cost difference in 1990 is that DRAM is 10 times more expensive.
- The cost difference in 1990 is that DRAM is 30 times more expensive.

a. [20] Which generation of DRAM chip—measured in bits per chip—will reach equity for each cost difference assumption? What year will that occur?

b. [10] What will be the difference in cost in the previous generation?

c. [Discussion] Do you think the cost difference in the previous generation is sufficient to prevent disks being replaced by DRAMs?

9.4 <9.2> [12/12/12] Assume a workload takes 100 seconds total, with the CPU taking 70 seconds and I/O taking 50 seconds.

a. [12] Assume that the floating-point unit is responsible for 25 seconds of the CPU time. You are considering a floating-point accelerator that goes five times faster. What is the time of the workload for maximum overlap, scaled overlap, and no overlap?

b. [12] Assume that seek and rotational delay of magnetic disks are responsible for 10 seconds of the I/O time. You are considering replacing the magnetic disks with solid state disks that will remove all the seek and rotational delay. What is the time of the workload for maximum overlap, scaled overlap, and no overlap?

c. [12] What is the time of the workload for scaled overlap if you make both changes?

9.5–9.9 Transaction-processing performance. The I/O bus and memory system of a computer are capable of sustaining 100 MB/sec without interfering with the performance of an 80-MIPS CPU (costing $50,000). Here are the assumptions about the software:

- Each transaction requires 2 disk reads plus 2 disk writes.
- The operating system uses 15,000 instructions for each disk read or write.
- The database software executes 40,000 instructions to process a transaction.
- The transfer size is 100 bytes.

You have a choice of two different types of disks:

- A 2.5-inch disk that stores 100 MB and costs $500.
- A 3.5-inch disk that stores 250 MB and costs $1250.
- Either disk in the system can support on average 30 disk reads or writes per second.

Answer the questions below using the TP-1 benchmark in Section 9.3. Assume that the requests are spread evenly to all the disks, that there is no waiting time due to busy disks, and that the account file must be large enough to handle 1000 TPS according to the benchmark ground rules.

9.5 <9.3,9.4> [20] How many TP-1 transactions per second are possible with each disk organization, assuming that each uses the minimum number of disks to hold the account file?

9.6 <9.3,9.4> [15] What is the system cost per transaction per second of each alternative for TP-1?

9.7 <9.3,9.4> [15] How fast a CPU makes the 100 MB/sec I/O bus a bottleneck for TP-1? (Assume that you can continue to add disks.)

9.8 <9.3,9.4> [15] As manager of MTP (Mega TP), you are deciding whether to spend your development money building a faster CPU or improve the performance of the software. The database group says they can reduce a transaction to 1 disk read and 1 disk write and cut the database instructions per transaction to 30,000. The hardware group can build a faster CPU that sells for the same amount of the slower CPU with the same development budget. (Assume you can add as many disks as needed to get higher performance.) How much faster does the CPU have to be to match the performance gain of the software improvement?

9.9 <9.3,9.4> [15/15] The MTP I/O group was listening at the door during the software presentation. They argue that advancing technology will allow CPUs to get faster without significant investment, but that the cost of the system will be dominated by disks if they don't develop new faster 2.5-inch disks. Assume the next CPU is 100% faster at the same cost and that the new disks have the same capacity as the old ones.

a. [15] Given the new CPU and the old software, what will be the cost of a system with enough old 2.5-inch disks so that they do not limit the TPS of the system ?

b. [15] Now assume you have as many new disks as you had old 2.5 inch disks in the original design. How fast must the new disks be (I/Os per second) to achieve the same TPS rate with the new CPU as the system in part a? What will the system cost?

9.10 <9.4> [20/20/20] Assume that we have the following two magnetic-disk configurations: a single disk and an array of four disks. Each disk has 20 surfaces, 885 tracks per surface with 16 sectors/track, each sector holds 1K bytes, and it revolves at 3600 RPM. Using the seek-time formula, for the IBM 3380D in Figure 9.39 (page 558). The time to switch between surfaces is the same as to move the arm one track. In the disk array all the spindles are synchronized—sector 0 in every disk rotates under the head at the exact same time—and the arms on all four disks are always over the same track. The data is "striped" across all 4 disks, so four consecutive sectors on a single disk system will be spread one sector per disk in the array. The delay of the disk controller is 2 ms per transaction, either for a single disk or for the array. Assume the performance of the I/O system is limited only by the disks and that there is a path to each disk in the array.

Compare the performance in both I/Os per second and megabytes per second of these two disk organizations assuming the following request patterns:

a. [20] Random reads of 4 KB of sequential sectors. Assume the 4 KB are aligned under the same arm on each disk in the array.

b. [20] Reads of 4 KB of sequential sectors where the average seek distance is 10 tracks. Assume the 4 KB are aligned under the same arm on each disk in the array.

c. [20] Random reads of 1 MB of sequential sectors. (If it matters, assume the disk controller allows the sectors to arrive in any order.)

9.11 [20] <9.4> Assume that we have one disk defined as in Exercise 9.9. Assume that we read the next sector after any read and that *all* read requests are one sector in length. We store the extra sectors that were read ahead in a *disk cache*. Assume that the probability of receiving a request for the sector we read ahead at some time in the future (before it must be discarded because the disk-cache buffer fills) is 0.1. Assume that we must still pay the controller overhead on a disk-cache read hit, and the transfer time for the disk cache is 250 ns per word. Is the read-ahead strategy faster? (Hint: Solve the problem in the steady state by assuming that the disk cache contains the appropriate information and a request has just missed.)

9.12–9.14 Assume the following information about our DLX machine:

Loads 2 cycles

Stores 2 cycles

All other instructions are 1 cycle. Use the summary instruction mix information in Figure C.4 in Appendix C on DLX for GCC.

Here are the cache statistics for a write-through cache:

- Each cache block is four words, and the whole block is read on any miss.
- Cache miss takes 13 cycles.
- Write through takes 6 cycles to complete, and there is no write buffer.

Here are the cache statistics for a write-back cache:

- Each cache block is four words, and the whole block is read on any miss.
- Cache miss takes 13 cycles for a clean block and 21 cycles for a dirty block.
- Assume that on a miss, 30% of the time the block is dirty.

Assume that the bus

- is only busy during transfers,
- transfers on average 1 word / clock cycle, and
- must read or write a single word at a time (it is not faster to read or write two at once).

9.12 [20/10/20/20] <9.4,9.5,9.6> Assume that DMA I/O can take place simultaneously with CPU cache hits. Also assume that the operating system can guarantee that there will be no stale-data problem in the cache due to I/O. The sector size is 1 KB.

a. [20] Assume the cache miss rate is 5%. On the average, what percentage of the bus is used for each cache write policy? This measured is called the *traffic ratio* in cache studies.

b. [10] If the bus can be loaded up to 80% of capacity without suffering severe performance penalties, how much memory bandwidth is available for I/O for each cache write policy? The cache miss rate is still 5%.

c. [20] Assume that a disk sector read takes 1000 clock cycles to initiate a read, 100,000 clock cycles to find the data on the disk, and 1000 clock cycles for the DMA to transfer the data to memory. How many disk reads can occur per million instructions executed for each write policy? How does this change if the cache miss rate is cut in half?

d. [20] Now you can have any number of disks. Assuming ideal scheduling of disk accesses, what is the maximum number of sector reads that can occur per million instructions executed?

9.13 [20/20] <9.4,9.5> Most machines today have a separate frame buffer to update the screen to avoid slowing down the memory system. An interesting issue is the percentage of the memory bandwidth that would be used if there were no frame buffer. Assume that all accesses to the memory are the size of a full cache block and they all take the time of a cache miss. The refresh rate is 60 Hz. Using the information in Section 9.4, calculate the memory traffic for the following graphics devices:

1. A 340 by 540 black-and-white display.
2. A 1280 by 1024 color display with 24 bits of color.
3. A 1280 by 1024 color display using a 256-word color map.

Assume the clock rate of the CPU is 60 MHz.

a. [20] What percentage of the memory/bus bandwidth do each of the three displays consume?

b. [20] Suppose instead of the bus and main memory being 32 bits wide that both are 512 bits wide. How long should a memory access take now using the wider bus? What percentage of memory bandwidth is now used by each display?

9.14 [20] <9.4,9.9> The IBM 3990 I/O Subsystem storage director can have a large cache for reads and writes. Assume the cache costs the same as four 3380D disks. What hit rate must the cache achieve to get the same performance as four more 3380D disks? (See Figure 9.15 (page 517) for 3380 performance.) Assume the cache could support 5000 I/Os per second if everything hit the cache.

9.15 [50] <9.3, 9.4> Take your favorite computer and write three programs that achieve the following:

1. Maximum bandwidth to and from disks

2. Maximum bandwidth to a frame buffer
3. Maximum bandwidth to and from the local area network

What is the percentage of the bandwidth that you achieve compared to what the I/O device manufacturer claims? Also record CPU utilization in each case for the programs running separately. Next run all three together and see what percentage of maximum bandwidth you achieve for three I/O devices as well as the CPU utilization. Try to determine why one gets a larger percentage than the others.

9.16 [40] <9.2> The system speedup formulas are limited to one or two types of devices. Derive simple to use formulas for unlimited numbers of devices, using as many different assumptions on overlap that you can handle.

9.17 [Discussion] <9.2> What are arguments for predicting system performance using maximum overlap, scaled overlap, and nonoverlap? Construct scenarios where each one seems most likely and other scenarios where each interpretation is nonsensical.

9.18 [Discussion] <9.11> What are the advantages and disadvantages of a minimal buffer I/O system like that used by IBM versus a maximal buffer I/O system on I/O system cost/performance?

The turning away from the conventional organization came in the middle 1960's, when the law of diminishing returns began to take effect in the effort to increase the operational speed of a computer. . . . Electronic circuits are ultimately limited in their speed of operation by the speed of light . . . and many of the circuits were already operating in the nanosecond range.

Bouknight et al. [1972]

. . . sequential computers are approaching a fundamental physical limit on their potential computational power. Such a limit is the speed of light . . .

A. L. DeCegama, *The Technology of Parallel Processing, Volume I* (1989)

... today's machines ... are nearing an impasse as technologies approach the speed of light. Even if the components of a sequential processor could be made to work this fast, the best that could be expected is no more than a few million instructions per second.

Mitchell [1989]

10 Future Directions

10.1 Introduction

In the first nine chapters we limited ourselves to ideas that have proven themselves in the marketplace. Yet the principles of these chapters can be found in the first paper on stored-program computers. The quotes on the facing page suggest that the days of the traditional computer are numbered. For a dated model of computation it has surely demonstrated its viability! Today it is improving in performance faster than at any time in its history, and the improvement in cost and performance since 1950 has been five orders of magnitude. Had the transportation industry kept pace with these advances, we could travel from San Francisco to New York in one minute for one dollar!

In this last chapter we abandon our conservative perspective and speculate about the future of computer architecture and compilers. The goal of innovative designs is dramatic improvements in cost/performance, or highly scalable performance with good cost/performance. Many of the ideas covered here have led to machines that are beginning to compete in the computer marketplace today. Some of them may not be around for the next edition of this book, while others may need their own chapters.

10.2 Flynn Classification of Computers

Flynn [1966] proposed a simple model of categorizing all computers. He looked at the parallelism in the instruction and data streams called for by the instructions at the most constrained component of the machine, and placed all computers in one of four categories:

1. *Single instruction stream, single data stream* (SISD, the uniprocessor)
2. *Single instruction stream, multiple data streams* (SIMD)
3. *Multiple instruction streams, single data stream* (MISD)
4. *Multiple instruction streams, multiple data streams* (MIMD)

This is a coarse model, as some machines are hybrids of these categories. Yet in this chapter we stick with this classic model because it is simple, easy to understand, gives a good first approximation, and—perhaps because of ease of understanding—is also the most widely used scheme.

Your first question about the model should be, "Single or multiple compared to what?" A machine that can add a 32-bit number in one clock cycle would seem to have multiple data streams when compared to a bit-serial computer that takes 32 clock cycles for the same operation. Flynn chose popular computers of that day, the IBM 704 and IBM 7090, as the model of SISD, although today any of the machines in Chapter 4 would serve as the example.

Having thus established the reference point for SISD, the next class is SIMD.

10.3 SIMD Computers—Single Instruction Stream, Multiple Data Streams

The cost of a general multiprocessor is, however, very high and further design options were considered which would decrease the cost without seriously degrading the power or efficiency of the system. The options consist of recentralizing one of the three major components.... Centralizing the [control unit] gives rise to the basic organization of [an]... array processor such as the Illiac IV.

Bouknight et al. [1972]

We have already seen typical instructions for a SIMD machine, yet the machine is not SIMD. The vector instructions of Chapter 7 operate on several data elements within a single instruction, executing in pipelined fashion in a single functional unit. Unlike SIMD, many functional units are not being invoked by a single instruction. A true SIMD would have, say, 64 data streams simultaneously going to 64 ALUs to form 64 sums within the same clock cycle.

The virtues of SIMD are that all the parallel execution units are synchronized and that they all respond to a single instruction from a single PC. From a programmer's perspective, this is close to the already familiar SISD. The original motivation for SIMD was to amortize the cost of the control unit over dozens of execution units. A more recently observed advantage is the reduced size of program memory—SIMD needs only one copy of the code being simultaneously executed, while MIMD needs a copy in every processor. Hence, the cost of program memory for a large number of execution units is less for SIMD.

Like vector machines, real SIMD computers have a mixture of SISD and SIMD instructions. There is a SISD host computer to perform operations such as branches or address calculation that do not need massive parallelism. The SIMD instructions are broadcast to all the execution units, each of which has its own set of registers. Also, as in vector machines, individual execution units can be disabled during a SIMD instruction. Unlike vector machines, massively parallel SIMD machines rely on interconnection or communication networks to exchange data between processing elements.

SIMD works best when vector instructions work best—in dealing with arrays in for-loops. Hence, to have the opportunity for massive parallelism in SIMD there must be massive amounts of data, or *data parallelism.* SIMD is at its weakest in case statements, where each execution unit must perform a different operation on its data, depending on what data it has. The execution units with the wrong data are disabled so that the proper units can continue. Such situations essentially run at 1/*n*th performance, where *n* is the number of cases.

The basic tradeoff in SIMD machines is performance of a processor versus number of processors. The machines in the marketplace today emphasize a large degree of parallelism over performance of the individual processors. The Connection Machine 2, for example, offers 65,536 single bit-wide processors while the ILLIAC IV had 64 64-bit processors.

While MISD fills out Flynn's classification, it is difficult to envision. A single instruction stream is simpler than multiple instruction streams, but multiple instruction streams with multiple data streams are easier to imagine than multiple instructions with a single data stream. A few of the architectures we have covered might be considered MISD: superscalar and VLIW architectures of Chapter 6 (Section 6.8) often have a single data stream and multiple instructions, although these machines have a single program counter. Perhaps closer to the mark are the decoupled architectures (pages 321–322), which have two instruction streams with independent program counters and a single data stream. Systolic architectures, covered in Section 10.6, might also be considered MISD.

While we can find examples of SIMD and MISD, their number is dwarfed by the multitude of MIMD machines.

10.4 MIMD Computers—Multiple Instruction Streams, Multiple Data Streams

> *Multis are a new class of computers based on multiple microprocessors. The small size, low cost, and high performance of microprocessors allow design and construction of computer structures that offer significant advantages in manufacture, price-performance ratio, and reliability over traditional computer families.... Multis are likely to be the basis for the next, the fifth, generation of computers.*
>
> Bell [1985, 463]

Practically since the first working computer, architects have been striving for the El Dorado of computer design: To compose a powerful computer by simply connecting many existing smaller ones. The user orders as many CPUs as he can afford and gets a commensurate amount of performance. Other advantages of MIMD may be highest absolute performance, faster than the largest uniprocessor, and highest reliability/availability (page 520) via redundancy.

For decades, computer designers have been looking for the missing piece of the puzzle that allows this speedup to happen, as if by magic. People are heard making statements that begin "Now that computers have dropped to such a low price..." or "This new interconnection scheme will overcome the scaling problem, so..." or "As this new programming language becomes widespread...," and end with "MIMDs will (finally) dominate computing."

With so many attempts to use parallelism, there are a few terms that are useful to know when discussing MIMDs. The principal division is that which delineates how information is shared. *Shared-memory* processors offer the programmer a single memory address that all processors can access; cache-coherent multiprocessors are shared-memory machines (see Sections 8.8 and 10.8). Processes communicate through shared variables in memory, with loads and stores capable of accessing any memory location. Synchronization must be available to coordinate processes. An alternative model to sharing data is where processes communicate by sending messages. As an extreme example, processes on different workstations communicate by sending messages over a local area network. This communication distinction is so fundamental that Bell suggests the term *multiprocessor* be limited to MIMDs that can communicate via shared memory, while MIMDs that can only communicate via explicit message passing should be called *multicomputers.* Since a portion of a shared memory could be used for messages, most multiprocessors can efficiently execute message-passing software. A multicomputer might be able to simulate shared memory by sending a message for every load or store, but presumably this would run excruciatingly slowly. Thus, Bell's distinction is based on the underlying hardware and program execution model, reflected in the performance of shared-memory communication, as opposed to the software that might run on a machine. Message-passing docents question the *scalability* of multiprocessors, while shared-memory advocates question the programmability of multicomputers. The next section examines this debate further.

The good news is that after many assaults, MIMD has established a beachhead. Today it is generally agreed that a multiprocessor may be more effective for a timesharing workload than a SISD. No single program takes less CPU time, but more independent tasks can be completed per hour—a throughput versus latency argument. Not only are start-up companies like Encore and Sequent selling small-scale multiprocessors, but the high-end machines from IBM, DEC, and Cray Research are multiprocessors. This means multiprocessors now embody a significant market, responsible for a majority of the mainframes and virtually all supercomputers. The only disappointment to computer architects is that shared memory is practically irrelevant for user programs run on the machine, with the operating system being the only benefactor. The development of a multiprocessor's operating system, particularly its resource manager, is simplified by shared memory.

The bad news is that it remains to be seen how many important applications run faster on MIMDs. The difficulty has not lain in the prices of SISDs, in flaws in topologies of interconnection networks, or in programming languages; but in the lack of applications software that have been reprogrammed to take advantage of many processors to complete important tasks sooner. Since it has been even harder to find applications that can take advantage of many processors, the challenge is greater for large scale MIMDs. When the positive gains from timesharing are combined with the scarcity of highly parallel applications, we can appreciate the predicament facing computer architects designing large-scale MIMDs that do not support timesharing.

But why is this so? Why should it be so much harder to develop MIMD programs than sequential programs? One reason is that it is hard to write MIMD programs that achieve close to linear speedup as the number of processors dedicated to the task increases. As an analogy, think of the communication overhead for a task done by one person versus the overhead for a task done by a committee, especially as the size of the group increases. While *n* people may have the potential to finish any task *n* times faster, the communication overhead for the group can prevent it from achieving this; this becomes especially hard as *n* increases. (Imagine the change in communication overhead going from 10 people to 1,000 people to 1,000,000.) Another reason for the difficulty in writing parallel programs is how much the programmer must know about the hardware. On a uniprocessor, the high-level language programmer writes his program ignoring the underlying machine organization—that's the job of the compiler. For a multiprocessor today, the programmer had better know the underlying hardware and organization if he is to write fast and scalable programs. This intimacy also makes portable parallel programs rare. Though this second obstacle may lessen over time, it is now the biggest challenge facing computer science. Finally, from Chapter 1 comes Amdahl's Law (page 8) to remind us that even small parts of a program must be parallelized to reach the full potential. Thus, coming close to linear speedup involves inventing new algorithms that are inherently parallel.

Example Suppose you want to achieve linear speedup with 100 processors. What fraction of the original computation can be sequential?

Answer Amdahl's Law is

$$\text{Speedup} = \frac{1}{(1-\text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

Substituting for the goal of linear speedup with 100 processors gives:

$$100 = \frac{1}{(1-\text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{100}}$$

Solving for percentage converted to enhanced mode:

$$100 - 100 * \text{Fraction}_{\text{enhanced}} + 1 * \text{Fraction}_{\text{enhanced}} = 1$$

$$-99 * \text{Fraction}_{\text{enhanced}} = -99$$

$$\text{Fraction}_{\text{enhanced}} = 1$$

Thus, to achieve linear speedup with 100 processors, **none** of the original computation can be sequential. Put another way, to get a speedup of 99 from 100 processors means the sequential fraction of the original program had to be about 0.0001.

The example above demonstrates the need for new algorithms. This underlines the authors' belief that major successes in using large-scale parallel machines of the 1990s are possible for those who understand applications, algorithms, and architecture.

10.5 The Roads to El Dorado

Figure 10.1 shows the state of the industry, plotting number of processors versus performance of an individual processor. The massive parallelism question is whether taking the high road or the low road in Figure 10.1 will get us to El Dorado. Currently we don't know enough about parallel programming and applications to be able to quantitatively trade-off number of processors versus performance per processor to achieve the best cost/performance.

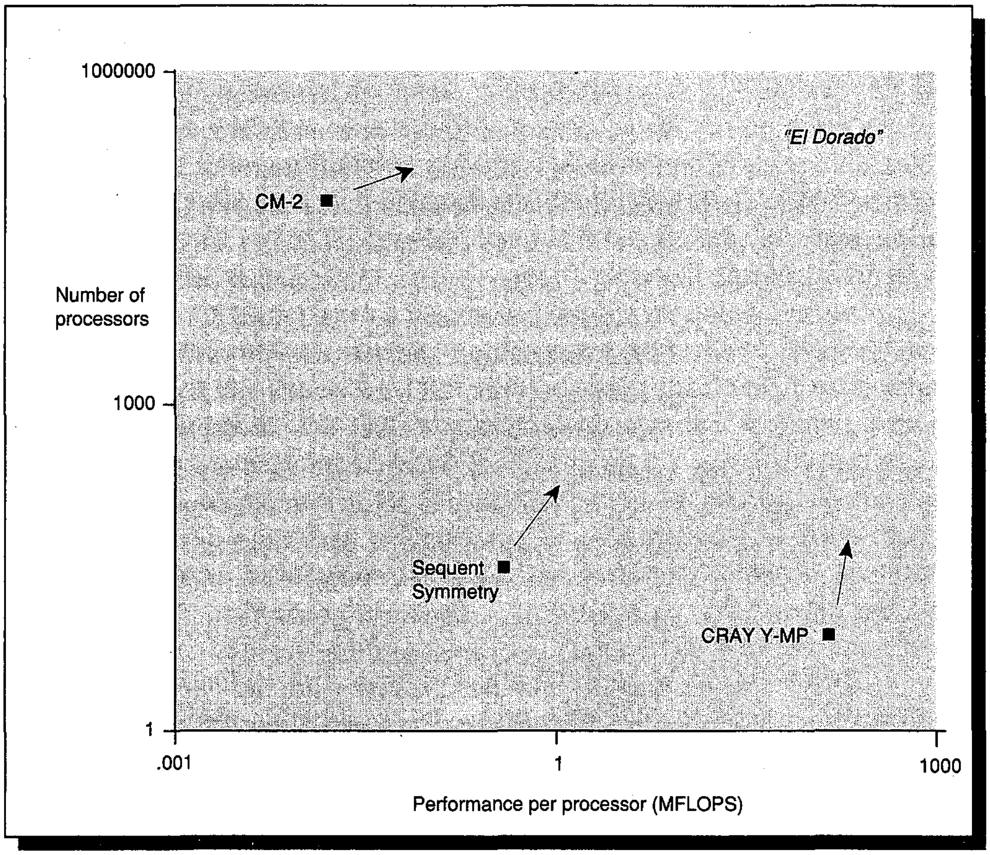


FIGURE 10.1 Danny Hillis, architect of the Connection Machines, has used a figure similar to this to illustrate the multiprocessor industry. (Hillis's x axis was processor width rather than processor performance.) Processor performance on this graph is approximated by the MFLOPS rating of a **single** processor for the DAXPY procedure of the Linpack benchmark for a 1000 x 1000 matrix. Generally, it is easier for programmers when moving to the right , while moving up is easier for the hardware designer because there is more hardware replication. The massive parallelism question is, "Which is the quickest path to the upper right corner?" The computer design question is, "Which has the best cost/performance or is more scalable for equivalent cost/performance?"

It is interesting to note that very different changes are required to improve performance depending on whether you take the low road or the high road in this figure. Since most programs are written in high-level languages, moving along the horizontal direction (increasing performance per processor) is almost entirely a matter of improving the hardware. The applications are unchanged, with compilers adapting them to the more powerful processor. Hence, increasing processor performance versus number of processors is easier for the applications software. Improving performance by moving in the vertical direction (increasing parallelism), on the other hand, may involve significant changes to applications, since programming ten processors may be very different from programming a thousand, and different yet again from programming a million. (But going from

100 to 101 is probably not different.) An advantage of the vertical path to performance is that the hardware may be simply replicated—the processors in particular, but also the hardware of the interconnection switch. Hence, increasing number of processors versus processor performance results in more hardware replication. An advantage of the low road is that it is much more likely that there will be a market at the various points along the way to El Dorado. In addition, those who take the high road must grapple with Amdahl's Law.

This brings us to a fundamental debate about the organization of memory in large-scale machines of the future. The debate unfortunately often centers on a false dichotomy: *shared memory* versus *distributed memory*. Shared memory means a single address space, implying implicit communication. The real opposite to a shared address is *multiple private address spaces,* implying explicit communication. Distributed memory refers to the location of the memory. If physical memory is divided into modules with some placed near each processor (which allows faster access time to that memory), then physical memory is distributed. The real opposite of distributed memory is *centralized memory,* where access time to a physical memory location is the same for all processors.

Clearly shared address versus multiple address and distributed memory versus centralized memory are orthogonal issues: SIMDs or MIMDs can have a shared address and a distributed physical memory or multiple private address spaces and a centralized physical memory (although this last combination would be unusual). Figure 10.2 categorizes several machines by these axes. The proper debates concerning the future are the pros and cons of a single address and the pros and cons of distributed memory.

The single address debate is closely tied to the model of communication, since shared-address machines must offer implicit communication (possibly

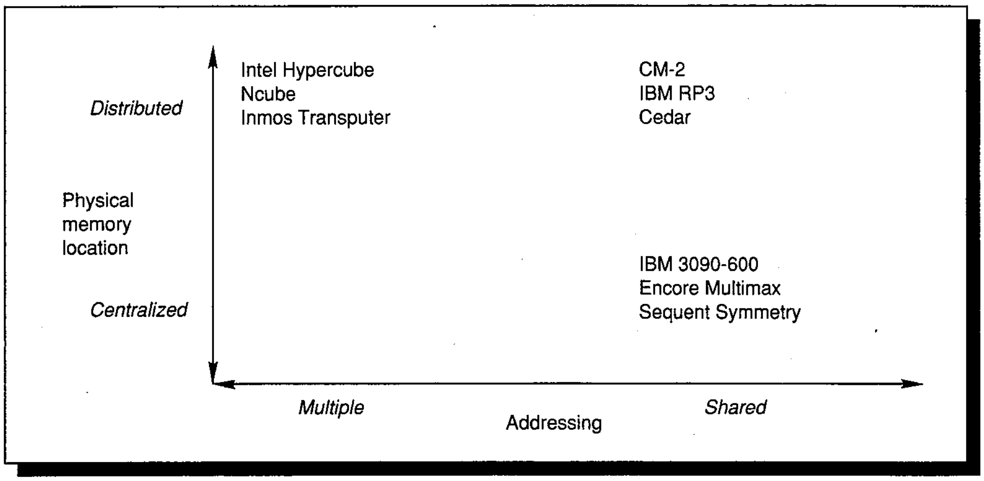


FIGURE 10.2 Parallel processors placed according to centralized versus distributed memory and shared versus multiple addressing. In general it is easier for software for machines on the shared side of the addressing axis and it is easier to build larger-scale machines on the distributed end of the vertical access. These machines in the graph are described in Section 10.11.

part of any memory access) and multiple-address machines must have explicit communication. (It is not quite that simple since some shared-address machines also offer explicit communication in various forms.) "Implicitists" knock "explicitists" for advocating machines that are harder to program when it is already hard to find applications: Why make the programmer's life more difficult when software is the linchpin of large-scale parallelism? One reply is that if memory is distributed, as processors get faster the time to remote memory will be so long—say 50 to 100 clock cycles—the compiler or programmer must be aware he is writing for a large-scale parallel machine no matter which communication scheme is used. Explicit communication also offers the possibility of hiding the cost of communication by overlapping it with computation. The implicitist reply is that using hardware rather than explicit instructions reduces the overhead of communication. Moreover, a single address means processes can use pointers and communicate data only if the pointer is dereferenced, while explicit communication means the data must be sent in the presence of pointers since the data **might** be accessed. The explicitist rebuttal is the owner of the data can send the data, traversing a properly designed network only once, while in shared-memory machines a processor requests the data and then the owner returns it, requiring two trips over the communications network.

Distributed-memory advocates argue that no matter how much caching is placed in front of a single central memory, it has limited bandwidth, and thus, limits the number of processors. Central-memory advocates raise the question of efficiency: If there is not enough parallelism to use many processors, then why distribute memory? Centralists also point out that distributed memory increases the difficulty of programming, since now the programmer or the compiler must decide how to lay out the data in the physical memory modules so as to reduce communication. Hence, distributed memory introduces the concept of data elements being near a processor (the module taking less time to access) or far (in other memory modules).

We can now explain a difficulty of the distributed versus centralized dichotomy. Every processor will likely have a cache, which is in some sense a distributed memory no matter how main memory is organized. Even with caches, the latency of a miss and the effective bandwidth for satisfying cache requests can be improved if data is allocated to the memory module near the appropriate cache. Hence, there is still a distinction between centralized and distributed main memory in the presence of caches.

As you can imagine, these debates continue back and forth, practically interminably. Fortunately, in computer architecture such disagreements are settled by measurements rather than polemics. Thus, time will be the judge of these issues, but your authors will be the judge of a bet inspired by these debates (see page 590 in 10.11).

The real issues for future machines are these: Do problems and algorithms with sufficient parallelism exist? And can people be trained or compilers be written to exploit such parallelism?

10.6 Special-Purpose Processors

In addition to exploring parallelism, many designers today are exploring special-purpose computers. With the increasing sophistication of *computer-aided design* software and increasing capacity per chip comes the opportunity of quickly building a chip that does one thing well at low cost. Real-time speech recognition and image processing are examples. Such special-purpose devices, or *coprocessors*, frequently act in conjunction with the CPU. There are two types in the coprocessor trend: digital signal processors and systolic arrays.

Digital signal processors (or DSPs) are not derived from the traditional model of computing, and tend to look like horizontal microprogrammed machines (see page 212) or VLIW machines (see pages 322–325). They tend to solve real-time problems, essentially having an infinite-input data stream. There has been little emphasis on compiling from programming languages such as C, but that is starting to change. As DSPs bend to the demands of programming languages, it will be interesting to see how they differ from traditional microprocessors.

Systolic arrays evolved from attempts to get more efficient computing bandwidth from silicon. Systolic arrays can be thought of as a method for designing special-purpose computers to balance resources, I/O bandwidth, and computation. Relying on pipelining, data flows in stages from memory through an array of computation units and back to memory, as suggested in Figure 10.3. Recently, systolic-array research has moved away from many, dedicated special-purpose chips to fewer, more powerful chips that are programmable.

The authors expect an increasing role for special-purpose computers in the 1990s because they offer both higher performance and lower cost for dedicated functions such as real-time speech recognition and image processing. The consumer marketplace seems the most likely candidate, given its high volume and sensitivity to cost.

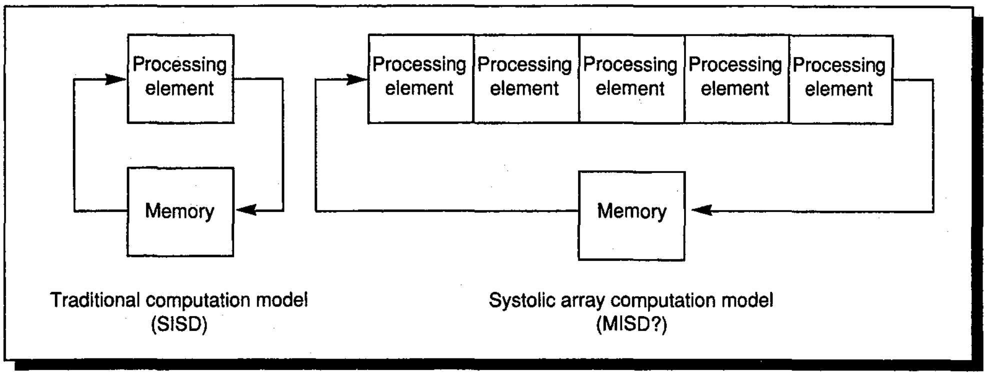


FIGURE 10.3 The systolic architecture gets its name from the heart rhythmically pumping blood. Data arrives at a processing element at regular intervals, where it is modified and passed to the next element, and so on, until it circulates back to memory. Some consider systolic arrays an example of MISD.

10.7 Future Directions for Compilers

Compilers of the future have two challenges on machines of the future:

- Lay out of data to reduce memory hierarchy and communication overhead, and
- Exploitation of parallelism.

Programs of the future will spend a larger percentage of the execution time waiting for the memory hierarchy as the gap grows between the clock cycle time of processors and the access time of main memory (see Figure 8.18, page 427). Compilers that arrange code and data so as to reduce cache misses may lead to larger performance improvements than traditional optimizations of today. Further improvements are possible with the possibility of prefetching data into a cache before it is needed by the program. One interesting proposition is by extending existing programming languages with array operations a programmer can express parallelism with calculations on entire arrays at a time, leaving it up to the compiler to lay out the data into processors to reduce the amount of communication. For example, the proposed extension to FORTRAN 77 called FORTRAN 8X includes array extensions. The hope is that the programmer's task might even be simpler than with SISD machines where array operations must be specified with loops. The range of programs that such a compiler can handle efficiently and the number of hints a programmer must supply on where to place data will determine the practical value of this proposal.

In addition to reducing the costs of memory access and communication, compilers may change performance by factors of two or three by utilizing parallelism available in the processor. Figure 2.25 (page 75) shows the Perfect Club benchmarks operate at only 1% of peak performance, clearly suggesting many opportunities for software. More specifically, the superscalar machines of Chapter 6 (pages 318–320) typically achieve a speedup of less than 2 using today's compilers, even through the potential performance improvement of executing 4 instructions at once is 4. From Chapter 7 we see that vector machines typically achieve a vectorization rate of 40% to 70%, delivering a speedup of 1.5 to 2.5, where a vectorization rate of 90% could achieve a speedup over 5. And current compilers for multiprocessors are considered successful if they achieve a speedup 3 for a single program when the potential from 8 processors is 8. Figure 10.4 (page 582) shows the potential improvement in performance of a larger percentage of the work executing in the higher-performance mode for each of these categories. Since we can expect multiple processors in machines where each processor has vector or superscalar features, the potential speedup of these factors may be multiplied together.

While this opportunity exists for compilers, we do not want to belittle its difficulty. Parallelizing compilers have been under development since 1975 but progress has been slow. These problems are hard, especially for the "dusty deck" challenge of running existing programs. Success has been limited to programs where the parallelism is available in the algorithm and expressed in the program and to machines with a small number of processors. Significant progress may eventually require new programming languages as well as smarter compilers!

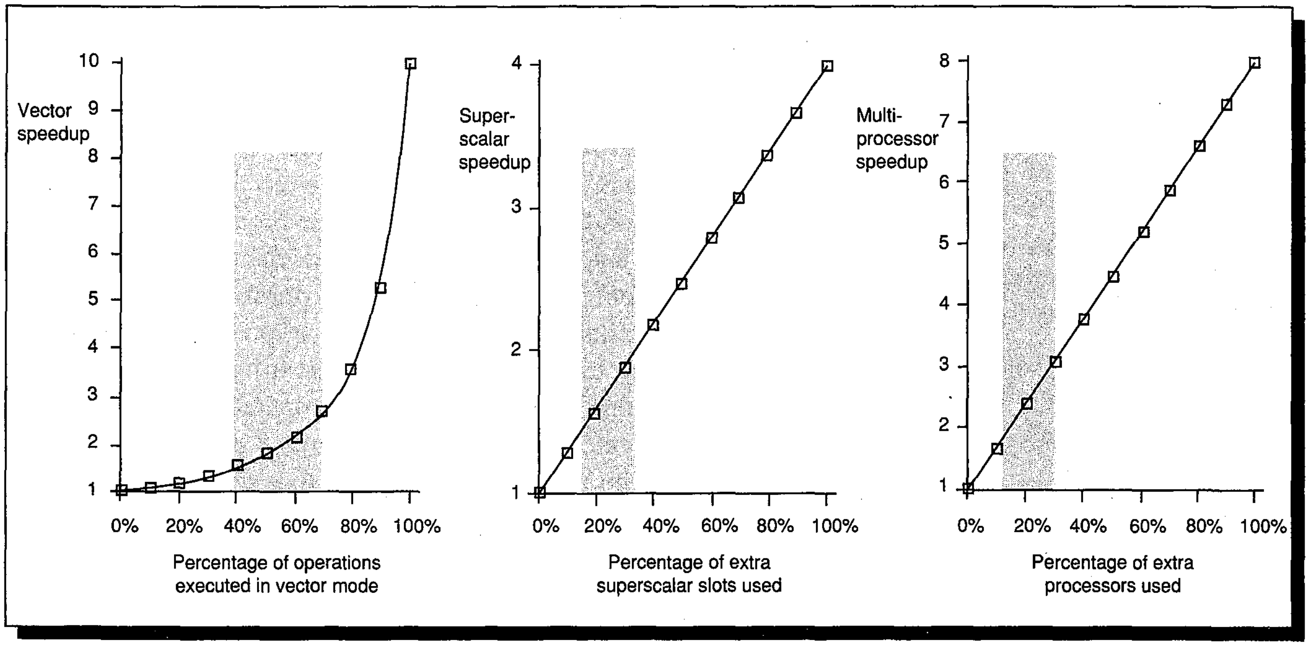


FIGURE 10.4 Potential for performance improvement by compilers transforming more of the computation into the faster mode. The leftmost graph shows the percentage of operations executed in vector mode, while the other graphs show the percentage of the potential speedup in use on average: percentage of four instructions used per cycle in superscalar and percentage of time all eight processors were utilized in the multiprocessor. The gray area shows the range of utilization typically found in programs using current compilers.

10.8 Putting It All Together: The Sequent Symmetry Multiprocessor

The high performance and low cost of the microprocessor inspired renewed interest in multiprocessors in the 1980s. Several microprocessors can be placed on a common bus because:

they are much smaller than multichip processors,

caches can lower bus traffic, and

coherency protocols can keep caches and memory consistent.

Traffic per processor and the bus bandwidth determine the number of processors in such a multiprocessor.

Several research projects and companies investigated these shared-bus multiprocessors. One example is Sequent Corporation, founded to build multiprocessors based on standard microprocessors, and the UNIX operating system. The first-generation system was the Balance 8000, offered in 1984 with 2 to 12 National 32032 microprocessors, a 32-bit split transaction bus that multiplexed address and data, and one 8-KB, 2-way–set-associative, write-through cache per processor. Each cache watched the bus to maintain coherency using write through with invalidate. (See Sections 8.4, 8.8, and 9.4 for a review of these terms.) The sustained bandwidth of the main memory and bus is 26.7 MB/sec. Two years later Sequent upgraded to the Balance 21000, offering up to 30 National 32032 microprocessors with the same memory system and bus.

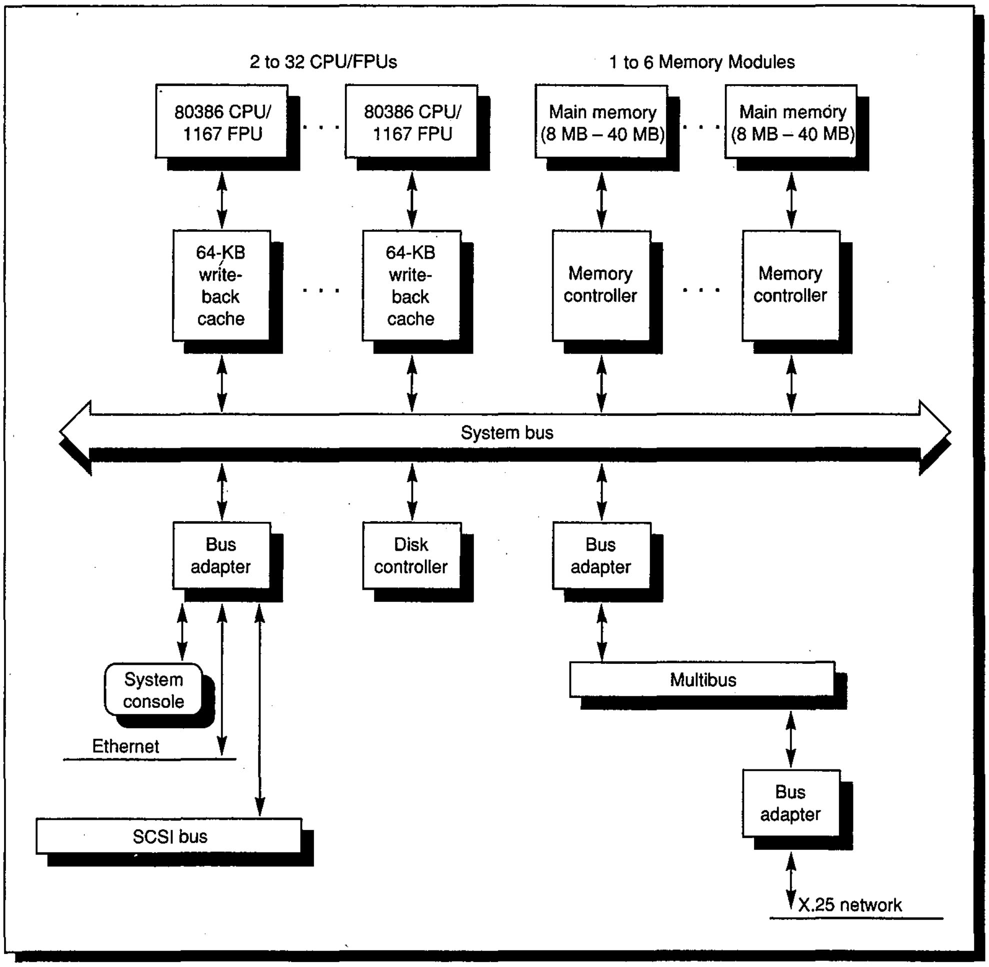


FIGURE 10.5 The Sequent Symmetry multiprocessor has up to 30 microprocessors, each with 64 KB of 2-way set associative, write-back caches connected over the shared system bus. Up to six memory controllers also talk to this 64-bit-wide bus, plus some interfaces for I/O. In addition to a special-purpose disk controller, there is an interface for the system console, Ethernet network, and SCSI I/O bus (see Chapter 9), as well as another interface for Multibus. I/O devices can be attached either to SCSI or to Multibus, as the customer desires. (Although all interfaces are labeled "Bus adapter," each is a unique design.)

In 1986, Sequent began the design of the Symmetry multiprocessor, assuming a microprocessor 300% to 400% faster than the 32032. The goal was to support as many processors as possible using the I/O controllers developed for the Balance system. This meant the bus had to remain compatible, though the new memory and bus system had to deliver roughly 300% to 400% higher bandwidth than the older system.

The goal of higher memory-system bandwidth with a similar bus was attacked on four levels. First, the cache was increased to 64 KB, increasing the hit rate and therefore the effective memory bandwidth as seen by the processor. Second, the cache policy was changed from write through to write back to reduce the number of write operations on the shared bus. To maintain cache coherency with write back, Symmetry uses a write-invalidate scheme (see pages 468–469). The third change was to double the bus width to 64 bits, thereby doubling the bus bandwidth to 53 MB/sec. The final change was to have each memory controller interleave memory as two banks (see Section 8.8), allowing the memory system to match the bandwidth of the wider bus. The memory system can have up to six controllers with up to 240-MB total main memory.

The use of high-level languages and the portability of the UNIX operating system allowed changing instruction sets to the faster Intel 80386. Running at a higher clock rate, with the faster Weitek 1167 floating-point accelerator, **and** with the improved memory system, a single 80386 ran from 214% to 776% faster for floating-point benchmarks and about 375% faster for integer benchmarks. Figure 10.5 (page 583) shows the organization of the Symmetry.

One of the other design constraints was that the new Symmetry boards had to work properly when put into the old Balance systems. Since the new system was to use write back and the old system used write through, the hardware team solved the problem by designing the new caches to support either write through or write back. Lovett and Thakkar [1988] took advantage of that feature to run parallel programs with both policies. Figure 10.6 shows bus utilization versus the number of processors for four parallel programs.

As mentioned above, bus utilization directly corresponds to the number of processors that can be used in such single-bus systems. Write-through caches should have higher bus utilization for the same number of processors since every write must go over the bus; or from a different perspective, the same bus should be able to support more processors if they use write-back caches. Figure 10.6 fulfills our expectations; the buses saturate with fewer than 16 processors with write through, but write back appears to scale to the full size.

There are two components to the bus traffic: normal misses and coherency support. Uniprocessor misses (compulsory, capacity, and conflict) can be reduced by larger caches and by better write policies, but the coherency traffic is a function of the parallel program. The primary benefit of write back for the programs in Figure 10.6 was simply reducing the number of writes on the bus due to the write-back policy, for there were few writes to shared data in these programs.

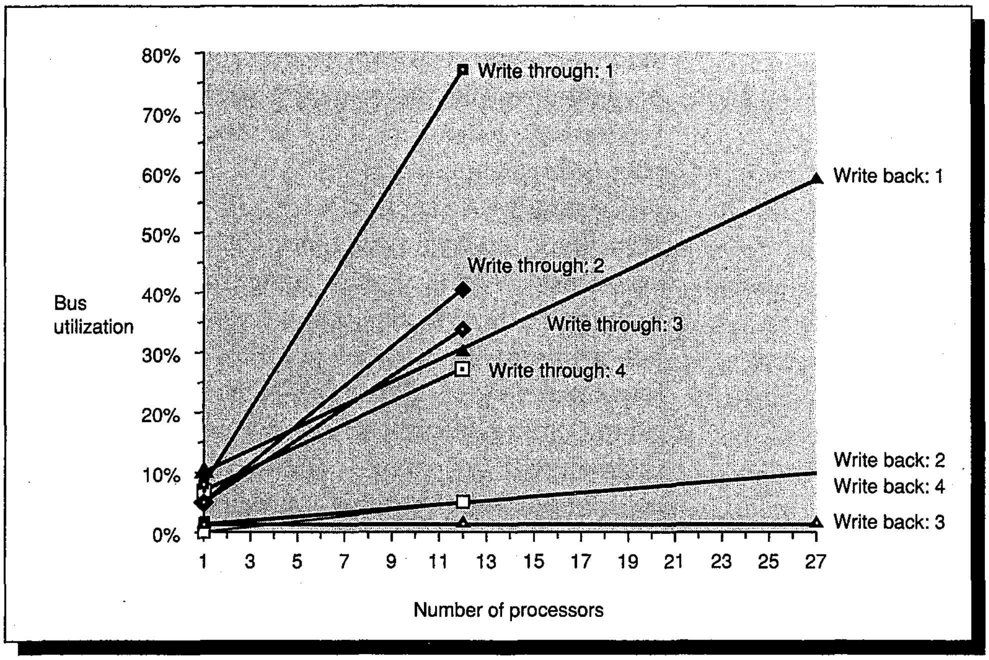


FIGURE 10.6 Comparing the impact of write-through versus write-back cache coherency on bus utilization of the Sequent Symmetry multiprocessor for four parallel benchmarks: (1) Butterfly Switch Simulator, (2) 2D Monte Carlo Simulation, (3) Ray Tracing , and (4) Parallel Linpack Benchmark. Lovett and Thakkar [1988] collected these data with a hardware performance monitor.

Another experiment evaluated the Symmetry as a timeshared (multiprogrammed) multiprocessor running ten independent programs. The experiment ran *n* copies of the program on *n* processors. This study found about half the programs started to stray from linearly increasing throughput at 6 to 8 processors with write through, yet with write back it stayed near linear for all but one of the ten programs for up to 28 processors. (The single dud was due to hot spots in the operating system rather than write-back coherency protocol.)

10.9 Fallacies and Pitfalls

Given the speculative nature of this chapter, it would seem that this section would not be needed. In good conscience, however, we submit two warnings.

Pitfall: Measuring performance of multiprocessors by linear speedup versus execution time.

"Mortar shot" graphs—plotting performance versus number of processors showing linear speedup, a plateau, and then a falling off—have long been used to judge the success of parallel processors. While scalability is one facet of a parallel program, it is not a direct measure of performance. The first question is the power of the processors being scaled: A program that linearly improves performance to equal 100 Intel 8080s may be slower than the sequential version on a workstation. Be especially careful of floating-point–intensive programs, as processing elements without hardware assist may scale wonderfully but have poor collective performance.

Comparing execution times is only fair if you are comparing the best algorithms on each machine. (Of course, you can't subtract time for idle processors when evaluating a multiprocessor, so CPU time is inappropriate for multiprocessors.) Comparing the identical code on two machines may seem fair, but it is not; the parallel program may be slower on a uniprocessor than a sequential version. Sometimes, developing a parallel program will lead to algorithmic improvements, so that comparing the previously best-known sequential program with the parallel code—which seems fair—will not compare equivalent algorithms. To reflect this issue, sometimes the terms *relative speedup* (same program) and *true speedup* (best programs) are used. Results that suggest *superlinear* performance, when a program on *n* processors is more than *n* times faster than the equivalent uniprocessor, give a clue to unfair comparisons.

Fallacy: Amdahl's Law doesn't apply to parallel computers.

In 1987, the head of a research organization claimed that Amdahl's Law (see Section 1.3) had been broken by a MIMD machine. This hardly meant, however, that the law has been overturned for parallel computers; the neglected portion of the program will still limit performance. To try to understand the basis of the media reports, let's see what Amdahl [1967] originally said:

> *A fairly obvious conclusion which can be drawn at this point is that the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude.* [page 483]

One interpretation of the law was that since portions of every program must be sequential, there is a limit to the useful economic number of processors—say 100. By showing linear speedup with 1000 processors, this interpretation of Amdahl's Law was disproved.

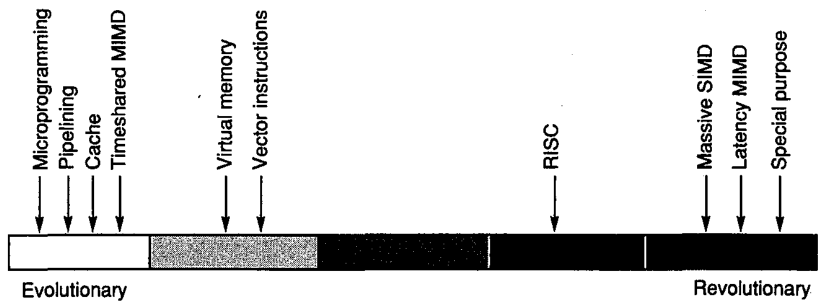
The approach of the researchers was to change the input to the benchmark, so that rather than going 1000 times faster, they essentially computed 1000 times more work in comparable time. For their algorithm the sequential portion of the program was constant independent of the size of the input, and the rest was fully parallel—hence, linear speedup with 1000 processors.

Chapter 2 (see Section 2.2) describes the dangers of letting each experimenter select his own input for benchmarks. We see no reason why varying input is safe for evaluating performance of multiprocessors, nor why Amdahl's Law doesn't apply. What this research does point out is the importance of having benchmarks that are large enough to demonstrate performance of large-scale parallel processors.

10.10 Concluding Remarks—Evolution Versus Revolution in Computer Architecture

Reading conference and journal articles from the last 20 years can leave one discouraged; so much effort has been expended with so little impact. Optimistically speaking, these papers act as gravel and, when placed logically together, form the foundation for the next generation of computers. From a more pessimistic point of view, if 90% of the ideas disappeared no one would notice.

One reason for this could be called the "von Neumann syndrome." By hoping to invent a new model of computation that will revolutionize computing, researchers are striving to become known as the von Neumann of the 21st century. Another reason is taste: researchers often select problems that no one else cares about. Even if important problems are selected, there is frequently a lack of experimental evidence to convincingly demonstrate the value of the solution. Moreover, when important problems are selected and the solutions are demonstrated, the proposed solutions may be too expensive relative to their



User compatibility	**Binary**	**Upward binary**	**Assembly**	**High-level language**	**New programs, extended or new HLL, new algorithms**
Example	VAX-11/780 vs. 8800	IBM 360 vs. 370 vs. 370-XA vs. ESA/370	MIPS 1000 vs. DECstation 3100	Sun 3 vs. Sun 4	SISD vs. CM-2
Difference	Microcode, TLB, caches, pipelining, MIMD	Some new instructions	Byte order (Big vs. Little Endian)	Full instruction set (same data representation)	Algorithms, extended HLL, programs

FIGURE 10.7 The evolution-revolution spectrum of computer architecture. The first four columns are distinguished from the last column in that applications and operating systems can be ported from other computers rather than written from scratch. For example, RISC is listed in the middle of the spectrum because user compatibility is only at the level of high-level languages, while microprogramming allows binary compatibility, and latency-oriented MIMDs require changes to algorithms and extending HLLs. Time-shared MIMD means MIMDs justified by running many independent programs at once, while latency MIMD means MIMDs intended to run a single program faster.

benefit. Sometimes this expense is measured as straightforward cost/performance—the performance enhancement does not merit the added cost. More often the expense of innovation is that it is too disruptive to computer users. Figure 10.7 shows what we mean by the *evolution-revolution spectrum* of computer architecture innovation. To the left are ideas that are invisible to the user (presumably excepting better cost, better performance, or both). This is the evolutionary end of the spectrum. At the other end are revolutionary architecture ideas. Those are the ideas that require new applications from programmers who must learn new programming languages and models of computation, and must invent new data structures and algorithms.

Revolutionary ideas are easier to publish than evolutionary ideas, but to be adopted they must have a much higher payoff. Caches are an example of an evolutionary improvement. Within five years after the first publication about caches almost every computer company was designing a machine with a cache. The RISC ideas were nearer to the middle of the spectrum, for it took closer to ten years for most companies to have a RISC product. An example of a revolutionary computer architecture is the Connection Machine. Every program that runs efficiently on that machine was either substantially modified or written especially for it, and programmers need to learn a new style of programming for it. Thinking Machines was founded in 1983, but only a few companies offer that style of machine.

There is value in projects that do not affect the computer industry because of lessons that they document for future efforts. The sin is not in having a novel architecture that is not a commercial success; the sin is in not quantitatively evaluating the strengths and weaknesses of the novel ideas. The next section mentions several machines whose primary contribution is documentation of the machine and experience using it.

When contemplating the future—and when inventing your own contributions to the field—remember the evolution-revolution spectrum. Also keep in mind the laws and principles of computer architecture found in the early chapters; these will surely guide computers of the future, just as they have guided computers of the past.

10.11 Historical Perspective and References

For over a decade prophets have voiced the contention that the organization of a single computer has reached its limits and that truly significant advances can be made only by interconnection of a multiplicity of computers in such a manner as to permit cooperative solution.... Demonstration is made of the continued validity of the single processor approach...

Amdahl [1967, 483]

The quotes at the chapter opening give the classic arguments for abandoning the current form of computing, and Amdahl [1967] gives the classic reply.

Arguments for the advantages of parallel execution can be traced back to 19th century [Menabrea 1842]! Yet the effectiveness of the multiprocessor for reducing latency of individual important programs is still being determined.

The earliest ideas on SIMD-style computers are from Unger [1958] and Slotnick, Borck, and McReynolds [1962]. Slotnick's Solomon design formed the basis of the Illiac IV, perhaps the most infamous of the supercomputer projects. While successful in pushing several technologies useful in later projects, it failed as a computer. Costs escalated from the $8 million estimate in 1966 to $31 million by 1972, despite constructing only a quarter of the planned machine. Actual performance was at best 15 MFLOPS versus initial predictions of 1000 MFLOPS for the full system (see Hord [1982]). Delivered to NASA Ames Research 1972, the computer took three more years of engineering before it was usable. These events slowed investigation of SIMD, with Danny Hillis [1985] resuscitating this style in the Connection Machine: The cost of a program memory for each of 65,636 1-bit processors was prohibitive, and SIMD was the solution.

It is difficult to distinguish the first multiprocessor. The first computer from the Eckert-Mauchly Corporation, for example, had duplicate units to improve availability. Holland [1959] gave early arguments for multiple processors. After several laboratory attempts at multiprocessors, the 1980s first saw successful commercial multiprocessors. Bell [1985] suggests the key was that the smaller size of the microprocessor allowed the memory bus to replace the interconnection network hardware, and that portable operating systems meant multiprocessor projects no longer required the invention of a new operating system. This is the paper in which he defines the terms "multiprocessor" and "multicomputer." Two of the best-documented multiprocessor projects are the C.mmp [Wulf and Bell 1972 and Wulf and Habrison 1978] and Cm* [Swan et al. 1977 and Gehringer, Siewiorek, and Segall 1987]. Recent commercial multiprocessors include the Encore Multimax [Wilson 1987] and the Sequent Symmetry [Lovett and Thakkar 1988]. The Cosmic Cube is an early multicomputer [Seitz 1985]. Recent commercial multicomputers are the Intel Hypercube and the Transputer-based machines [Whitby-Strevens 1985]. Attempts at building a scalable shared-memory multiprocessor include the IBM RP3 [Pfister, Brantley, George, Harvey, Kleinfekder, McAuliffe, Melton, Norton, and Weiss 1985], the NYU Ultracomputer [Schwartz 1980 and Elder, Gottlieb, Kruskal, McAuliffe, Randolph, Snir, Teller, and Wilson 1985], and the University of Illinois Cedar project [Gajksi, Kuck, Lawrie, and Sameh 1983].

There is unbounded information on multiprocessors and multicomputers: Conferences, journal papers, and even books seem to be appearing faster than any single person can absorb the ideas. One good source is the International Conference on Parallel Processing, which has met annually since 1972. Two recent books on parallel computing have been written by Almasi and Gottlieb [1989] and Hockney and Jesshope [1988]. Eugene Miya of NASA Ames has collected an on-line bibliography of parallel-processing papers that contains more than 10,000 entries. To highlight a few papers, he sends out electronic requests every January to ask which papers every serious student in the field should read. After collecting the ballots, he picks the ten papers most frequently recommended and publishes that list. Here is an alphabetical list of the winners: Andrews and Schneider [1983]; Batcher [1974]; Dewitt, Finkel, and Solomon [1984]; Kuhn and Padua [1981]; Lipovski and Tripathi [1977]; Russell [1978]; Seitz [1985]; Swan, Fuller, and Siewiorek [1977]; Treleaven, Brownbridge, and Hopkins [1982]; and Wulf and Bell [1972].

Special-purpose computers predate the stored-program computer. Brodersen [1989] gives a history of signal processing and its evolution to programmable devices. H. T. Kung [1982] coined the term "systolic array" and has been one of the leading proponents of this style of computer design. Recent research has been in the direction of making programmable systolic-array elements and providing a programming environment to simplify the programming task.

Its hard to predict the future, yet Gordon Bell has made two predictions for 1995. The first is that a computer capable of sustaining a TeraFLOPS—one million MFLOPS—will be constructed by 1995, either using a multicomputer with 4K to 32K nodes or a Connection Machine with several million processing elements [Bell 1989]. To put this prediction in perspective, each year the Gordon Bell Prize acknowledges advances in parallelism, including the fastest real program (highest MFLOPS). In 1988, the winner achieved 400 MFLOPS using a CRAY X-MP with four processors and 16 megawords and in 1989 the winner used an eight-processor CRAY Y-MP to run at 1680 MFLOPS. Machines and programs will have to improve by a factor of three each year for the fastest program to achieve 1 TFLOPS in 1995.

The second Bell prediction concerns the number of data streams in supercomputers shipped in 1995. Danny Hillis believes that while supercomputers with a small number of data streams may be best sellers, the biggest machines will be machines with many data streams, and these will perform the bulk of the computations. Bell bet Hillis that in the last quarter of calendar year 1995 more sustained MFLOPS will be shipped in machines using few data streams (≤100) rather than many data streams (≥1000). This bet concerns only supercomputers, defined as machines costing more than $1,000,000 and used for scientific applications. Sustained MFLOPS is defined for this bet as the number of floating-point operations per **month**, so availability of machines affects their rating. The loser must write and publish an article explaining why his prediction failed; your authors will act as judge and jury.

References

ALMASI, G. S. AND A. GOTTLIEB [1989]. *Highly Parallel Computing,* Benjamin/Cummings, Redwood City, Calif.

AMDAHL, G. M. [1967]. "Validity of the single processor approach to achieving large scale computing capabilities," *Proc. AFIPS Spring Joint Computer Conf.* 30, Atlantic City, N. J. (April) 483–485.

ANDREWS, G. R. AND F. B. SCHNEIDER [1983]. "Concept and notations for concurrent programming," *Computing Surveys* 15:1 (March) 3–43.

BATCHER, K. E. [1974]. "STARAN parallel processor system hardware," *Proc. AFIPS National Computer Conference*, 405–410.

BELL, C. G. [1985]. "Multis: A new class of multiprocessor computers," *Science* 228 (April 26) 462–467.

BELL, C. G. [1989]. "The future of high performance computers in science and engineering," *Comm. ACM* 32:9 (September) 1091–1101.

BOUKNIGHT, W. J, S. A. DENEBERG, D. E. MCINTYRE, J. M. RANDALL, A. H. SAMEH, AND D. L. SLOTNICK [1972]. "The ILLIAC IV system," *Proc. IEEE* 60:4, 369–379. Also appears in D. P. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples* (1982), 306–316.

BRODERSEN, R. W. [1989]. "Evolution of VLSI signal-processing circuits," *Proc. Decennial Caltech Conf. on VLSI* (March) 43–46, The MIT Press, Pasadena, Calif.

DEWITT, D. J., R. FINKEL, AND M. SOLOMON [1984]. "The CRYSTAL multicomputer: Design and implementation experience, Computer Sciences Tech. Rep. No. 553, University of Wisconsin-Madison, September.

ELDER, J., A. GOTTLIEB, C. K. KRUSKAL, K. P. MCAULIFFE, L. RANDOLPH, M. SNIR, P. TELLER, AND J. WILSON [1985]. "Issues related to MIMD shared-memory computers: The NYU Ultracomputer approach," *Proc. 12th Int'l Symposium on Computer Architecture* (June), Boston, Mass., 126–135.

FLYNN, M. J. [1966]. "Very high-speed computing systems," *Proc. IEEE* 54:12 (December) 1901–1909.

GAJSKI, D., D. KUCK, D. LAWRIE, AND A. SAMEH [1983]. "CEDAR—A large scale multiprocessor," *Proc. Int'l Conf. on Parallel Processing* (August) 524–529.

GEHRINGER, E. F., D. P. SIEWIOREK, AND Z. SEGALL [1987]. *Parallel Processing: The Cm* Experience*, Digital Press, Bedford, Mass.

HILLIS, W. D. [1985]. *The Connection Machine,* The MIT Press, Cambridge, Mass.

HOCKNEY, R. W. AND C. R. JESSHOPE [1988]. *Parallel Computers-2, Architectures, Programming and Algorithms,* Adam Hilger Ltd., Bristol, England and Philadelphia.

HOLLAND, J. H. [1959]. "A universal computer capable of executing an arbitrary number of subprograms simultaneously," *Proc. East Joint Computer Conf.* 16, 108–113.

HORD, R. M. [1982]. *The Illiac-IV, The First Supercomputer,* Computer Science Press, Rockville, Md.

KUHN, R. H. AND D. A. PADUA, EDS. [1981]. *Tutorial on Parallel Processing*, IEEE.

KUNG, H. T. [1982]. "Why systolic architectures?," *IEEE Computer* 15:1, 37–46.

LIPOVSKI , A. G. AND A. TRIPATHI [1977]. "A reconfigurable varistructure array processor," *Proc. 1977 Int'l Conf. of Parallel Processing* (August), 165–174.

LOVETT, T. AND S. THAKKAR [1988]. "The Symmetry multiprocessor system," *Proc. 1988 Int'l Conf. of Parallel Processing*, University Park, Pennsylvania, 303–310.

MENABREA, L. F. [1842]. "Sketch of the analytical engine invented by Charles Babbage," Bibiothèque Universelle de Genève (October).

MITCHELL, D. [1989]. "The Transputer: The time is now," *Computer Design*, RISC supplement, 40–41 (November).

PFISTER, G. F., W. C. BRANTLEY, D. A. GEORGE, S. L. HARVEY, W. J. KLEINFEKDER, K. P. MCAULIFFE, E. A. MELTON, V. A. NORTON, AND J. WEISS [1985]. "The IBM research parallel processor prototype (RP3): Introduction and architecture," *Proc. 12th Int'l Symposium on Computer Architecture* (June), Boston, Mass., 764–771.

RUSSELL, R. M. [1978]. "The Cray-1 computer system," *Comm. ACM* 21:1 (January) 63–72.

SEITZ, C. [1985]. "The Cosmic Cube," *Comm. ACM* 28:1 (January) 22–31.

SLOTNICK, D. L., W. C. BORCK, AND R. C. MCREYNOLDS [1962]. "The Solomon computer," *Proc. Fall Joint Computer Conf.* (December), Philadelphia, 97–107.

SWAN, R. J., A. BECHTOLSHEIM, K. W. LAI, AND J. K. OUSTERHOUT [1977]. "The implementation of the Cm* multi-microprocessor," *Proc. AFIPS National Computing Conf.*, 645–654.

SWAN, R. J., S. H. FULLER, AND D. P. SIEWIOREK [1977]. "Cm*—A modular, multi-microprocessor," *Proc. AFIPS National Computer Conf.* 46, 637–644.

SWARTZ, J. T. [1980]. "Ultracomputers," *ACM Transactions on Programming Languages and Systems* 4:2, 484–521

TRELEAVEN, P. C., D. R. BROWNBRIDGE, and R. P. HOPKINS [1982]. "Data-driven and demand-driven computer architectures," *Computing Surveys*, 14:1 (March) 93–143.

UNGER, S. H. [1958]. "A computer oriented towards spatial problems," *Proc. Institute of Radio Engineers* 46:10 (October) 1744–1750.

VON NEUMANN, J. [1945]. "First draft of a report on the EDVAC." Reprinted in W. Aspray and A. Burks, eds., *Papers of John von Neumann on Computing and Computer Theory* (1987), 17–82, The MIT Press, Cambridge, Mass.

WHITBY-STREVENS C. [1985]. "The transputer," *Proc. 12th Int'l Symposium on Computer Architecture*, Boston, Mass. (June) 292–300.

WILSON, A. W., JR. [1987]. "Hierarchical cache/bus architecture for shared memory multiprocessors," *Proc. 14th Int'l Symposium on Computer Architecture* (June), Pittsburg, Penn., 244–252.

WULF, W. AND C. G. BELL [1972]. "C.mmp—A multi-mini-processor," *Proc. AFIPS Fall Joint Computing Conf.* 41, part 2, 765–777.

WULF, W. AND S. P. HARBISON [1978]. "Reflections in a pool of processors—An experience report on C.mmp/Hydra," *Proc. AFIPS 1978 National Computing Conf.* 48 (June), Anaheim, Calif. 939–951.

EXERCISES

10.1 [Discussion] <10.4> The weakness of SIMD for case statements, as well as the failure of the first machine to popularize SIMD, prevented exploration of SIMD designs while MIMD was still an open frontier. MIMD also has the advantage of riding the wave of improvements in SISD processors. Now that MIMD programming has not succumbed easily to assaults of computer scientists, the issue arises whether the simpler programming model of SIMD might lead it to victory over MIMD for large numbers of processors. It looks as if MIMD programs for thousands of processors will consist of thousands of copies of one program rather than thousands of different programs. Thus, the direction is toward a single **program** with multiple data streams, independent of whether the machine itself is SIMD or MIMD. What trends favor MIMD over SIMD, and vice versa? Be sure to consider utilization of memory and processors (including communication and synchronization).

10.2 [Discussion] <10.3–10.5> It might take approximately 100 clocks to communicate in a massively parallel SIMD or MIMD machine. What hardware techniques might this time? How can you change the architecture or the programming model to make a computer more immune to such delays?

10.3 [Discussion] <10.4,10.8> What must happen before latency-oriented MIMD machines become commonplace?

10.4 [Discussion] <10.6> When do special-purpose processors make sense economically?

10.5 [Discussion] <10.8> Construct a scenario whereby a truly revolutionary architecture—pick your favorite candidate—will play a significant role. Significant is defined as 10% of the computers sold, 10% of the users, 10% of the money spent on computers, or 10% of some other figure of merit.

10.6 [30] <10.2> The CM-2 uses 64K 1-bit processors in SIMD mode. Bit-serial operations can easily be simulated 32 bits one step by a 32-bit-wide SISD, at least for logical operations. The CM-2 takes about 500 ns for such operations. If you have access to a fast SISD, calculate how long add and logical AND take on 64K 1-bit numbers.

10.7 [30] <10.2> Similar to the question above, a popular use of the CM-2 is to operate on 32-bit data using multiple steps with the 64K 1-bit processors. The CM-2 takes about 16 microseconds for a 32-bit AND or add. Simulate this activity on a fast SISD; calculate how long it takes to add and logical AND 64K 32-bit numbers.

10.8–10.12 <2.2,10.4> **If you have access to a few different multiprocessors or multicomputers, performance comparison is the basis of some projects.**

10.8 [50] <2.2,10.4> One argument for super-linear speedup (pages 585–586) is that time spent servicing interrupts or switching contexts is reduced when you have many processors, since only one need service interrupts and there are more processors to be shared by users. Measure the time spent on a workload in handling interrupts or context switching on a uniprocessor versus a multiprocessor. This workload may be a mix of independent jobs for a multiprogramming environment or a single large job. Does the argument hold?

10.9 [50] <2.2,10.4> A multiprocessor or multicomputer is typically marketed using programs that can scale performance linearly with the number of processors. The project would be to port programs written for one machine to the others and measure their absolute performance and how it changes as you change the number of processors. What changes need to be made to improve performance of the ported programs on each machine? What is the ratio of processor performance according to each program?

10.10 [50] <2.2,10.4> Instead of trying to create fair benchmarks, invent programs that make one multiprocessor or multicomputer look terrible compared to the others, and also programs that always make one look better than the others. It would be an interesting result if you couldn't find a program that made one multiprocessor or multicomputer look worse than the others. What are the key performance characteristics of each organization?

10.11 [50] <2.2,10.4> Multiprocessors and multicomputers usually show performance increases as you increase the number of processors, with the ideal being *n* times speedup for *n* processors. The goal of this biased benchmark is to make a program that gets worse performance as you add processors. For example, this means that 1 processor on the multiprocessor or multicomputer runs the program fastest, 2 is slower, 4 is slower than 2, and so on. What are the key performance characteristics for each organization that give inverse linear speedup?

10.12 [50] <10.4> Networked workstations can be considered multicomputers, albeit with slow communication relative to computation. Port multicomputer benchmarks to a network using remote procedure calls for communication. How well do the benchmarks scale on the network versus the multicomputer? What are the practical differences between networked workstations and a commercial multicomputer?

The Fast drives out the Slow even if the Fast is wrong.

W. Kahan

by David Goldberg
(Xerox Palo Alto Research Center)

A Computer Arithmetic

A.1 Introduction

A tremendous variety of algorithms have been proposed for use in floating-point accelerators. However, actual floating-point chips are usually based on refinements and variations of just a few basic algorithms. In this appendix, we focus on those algorithms. In addition to choosing algorithms for addition, subtraction, multiplication and division, the computer architect must decide whether to go beyond the basics. Should square root be implemented in hardware or software? Should extended precision be implemented? This appendix will give you the background for making these and other decisions.

Our discussion of floating point will focus almost exclusively on the IEEE floating-point standard (IEEE 754) because of its rapidly increasing acceptance. Although floating-point arithmetic involves manipulating exponents and shifting fractions, the bulk of the time in floating-point operations is spent operating on fractions using integer algorithms (but not necessarily using the integer hardware). Thus, after our discussion of floating point, we will take a more detailed look at integer algorithms.

Some good references on computer arithmetic, in order from least to most detailed, are Chapter 7 of Hamacher, Vranesic, and Zaky [1984], Gosling [1980], and Scott [1985].

A.2 Basic Techniques of Integer Arithmetic

Readers who have studied computer arithmetic before will find most of this section to be review.

Ripple-Carry Addition

The building blocks of an adder that can compute the sum of the n-bit numbers $a_{n-1} \cdots a_1 a_0$ and $b_{n-1} \cdots b_1 b_0$ are *half adders* and *full adders*. The half adder takes two bits a_i and b_i as input and produces a sum bit s_i and a carry bit c_{i+1} as output. Mathematically, $s_i = (a_i + b_i) \bmod 2$, and $c_{i+1} = \lfloor (a_i + b_i)/2 \rfloor$, where $\lfloor \ \rfloor$ is the floor function. As logic equations, $s_i = a_i \overline{b_i} + \overline{a_i} b_i$, and $c_{i+1} = a_i b_i$, where $a_i b_i$ means $a_i \wedge b_i$ and $a_i + b_i$ means $a_i \vee b_i$. The half adder is also called a (2,2) adder, since it takes two inputs and produces two outputs. The full adder is a (3,2) adder and is defined by the logic equations

$$s_i = a_i \overline{b_i} \overline{c_i} + \overline{a_i} b_i \overline{c_i} + \overline{a_i} \overline{b_i} c_i + a_i b_i c_i \tag{A.2.1}$$

$$c_{i+1} = a_i b_i + a_i c_i + b_i c_i \tag{A.2.2}$$

The input c_i is called the *carry in*, while c_{i+1} is the *carry out*. The principle problem in building an adder for n-bit numbers is propagating the carries. The most obvious way to solve this is with a *ripple-carry adder*, consisting of n full adders, as illustrated in Figure A.1. (In the figures in this appendix the least significant bit is always on the right.) The c_{i+1} output of the ith adder is fed into the c_{i+1} input of the next adder (the $(i + 1)$-th adder) with the lower order carry in c_0 set to 0. Since the low-order carry in is zero, the low-order adder could be a half adder. Later, however, we will see that setting the low-order carry-in bit to 1 is useful for performing subtraction.

From Equation A.2.2, there are two levels of logic involved in computing c_{i+1} from c_i. Thus, if the least significant bit generates a carry, and that carry gets propagated all the way to the last adder, the a_0 signal will pass through $2n$ levels of logic before the final gate can determine whether there is a carry out of the most significant place. In general, the time a circuit takes to produce an output is proportional to the maximum number of logic levels through which a signal travels. However, determining the exact relationship between logic levels and timings is highly technology dependent. Therefore, when comparing adders we will simply compare the number of logic levels in each one. For a ripple-carry adder that operates on n bits, there are $2n$ logic levels. Typical values of n are 32 for integer arithmetic and 53 for double-precision floating point. The ripple-carry adder is the slowest adder, but also the cheapest. It can be built with only n simple cells, connected in a simple, regular way.

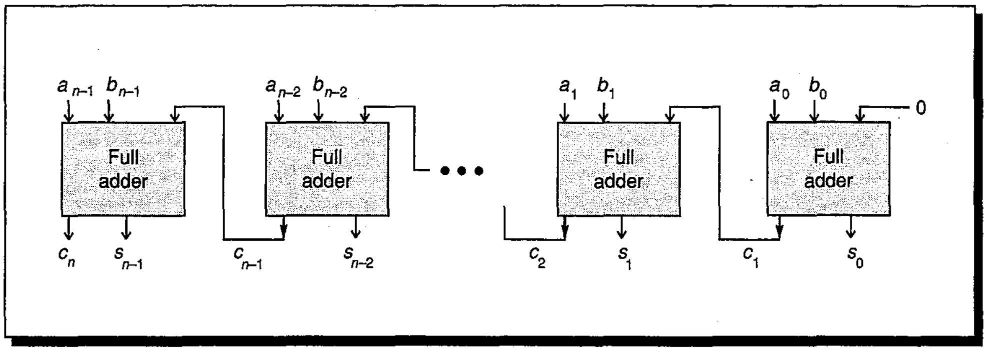


FIGURE A.1 Ripple-carry adder, consisting of *n* full adders. The carry out of one full adder is connected to the carry in of the adder for the next most significant bit. The carries ripple from the least significant bit (on the right) to the most significant bit (on the left).

Because the ripple-carry adder is relatively slow compared to the designs discussed in Section A.8, one might wonder why it is used at all. In technologies like CMOS, even though ripple adders take time O(*n*), the constant factor is very small. In such cases short ripple adders are often used as building blocks in larger adders.

Radix-2 Multiplication and Division

The simplest multiplier operates on two unsigned numbers, one bit at a time, as illustrated in Figure A.2(a) (page A-4). The numbers to be multiplied are $a_{n-1}a_{n-2}\cdots a_0$ and $b_{n-1}b_{n-2}\cdots b_0$, and they are placed in registers A and B, respectively. Register P is initially zero. There are two parts in each multiply step.

1. If the least significant bit of A is 1, then register B, containing $b_{n-1}b_{n-2}\cdots b_0$, is added to P; otherwise 00···00 is added to P. The sum is placed back into P.
2. Registers P and A are shifted right, with the low-order bit of P being moved into register A and the rightmost bit of A, which is not used in the rest of the algorithm, being shifted out.

After *n* steps, the product appears in registers P and A, with A holding the lower-order bits.

The simplest divider also operates on unsigned numbers and produces a bit at a time. A hardware divider is shown in Figure A.2(b). To compute *a*/*b*, put *a* in the A register, *b* in the B register, 0 in the P register, and then proceed as follows:

1. Shift the register pair (P,A) one bit left.
2. Subtract the content of register B (which is $b_{n-1}b_{n-2}\cdots b_0$) from register P.
3. If the result of step 2 is negative, set the low-order bit of A to 0, otherwise to 1.

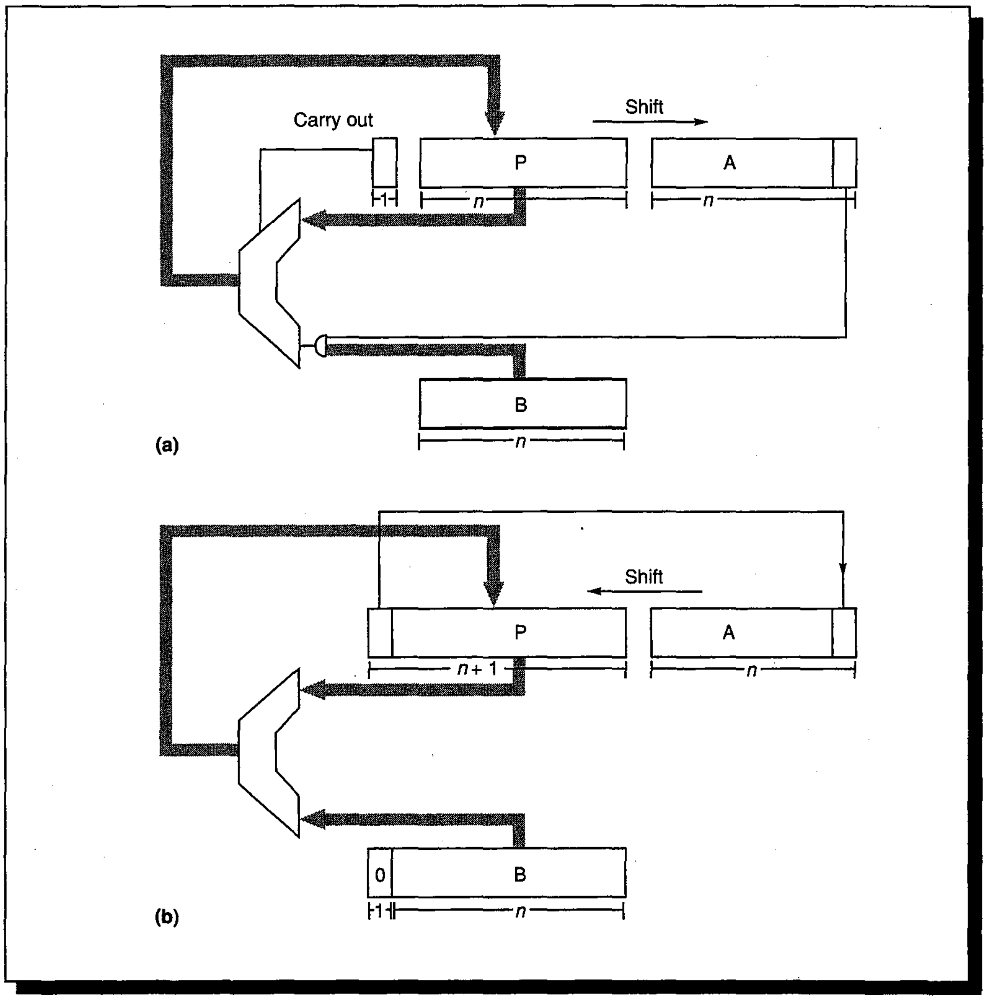


FIGURE A.2 Block diagram of simple multiplier (a) and divider (b) for *n*-bit unsigned integers. Each multiplication step consists of adding the contents of P to either B or 0 (depending on the low-order bit of A), replacing P with the sum, and then shifting both P and A one bit right. Each division step involves first shifting P and A one bit left, subtracting B from P, and if the difference is nonnegative, putting it into P. If the difference is nonnegative, the low-order bit of A is set to 1.

4. If the result of step 2 is negative, restore the old value of P by adding the contents of register B back into P.

After repeating this *n* times, the A register will contain the quotient, and the P register will contain the remainder. This algorithm is the binary version of the paper-and-pencil method; a numerical example is illustrated in Figure A.3(a) (page A-6).

Notice that the two block diagrams in Figure A.2 are very similar. The main difference is that the register pair (P,A) shifts right when multiplying and left when dividing. By allowing these registers to shift bidirectionally, the same hardware can be shared between multiplication and division.

The division algorithm illustrated in Figure A.3(a) (page A-6) is called *restoring*, because if subtraction by b yields a negative result, the P register is restored by adding b back in. The restoration step (4 above) can be easily eliminated. To see why, let r be the contents of the (P,A) register pair, with a binary point between the low-order bit of P and the high-order bit of A. Then each step of the algorithm computes $2r - b$, putting the high-order word of this difference in P, and the low-order word in A. Suppose the result of a step is negative. Normally, we would add b back in (giving $2r$), shift (giving $4r$), and then subtract (obtaining $4r - b$). Suppose we didn't restore, but continued with the algorithm. First, shift the unrestored $2r - b$, yielding $4r - 2b$, then add b, giving $4r - b$. This is exactly what we would have obtained if we had restored! Thus, the *nonrestoring* algorithm is

If P is negative,

1a. Shift the register pair (P,A) one bit left.

2a. Add the contents of register B to P.

Else,

1b. Shift the register pair (P,A) one bit left.

2b. Subtract the contents of register B from P.

Finally,

3. If P is negative, set the low-order bit of A to 0, otherwise set it to 1.

After repeating this n times, the quotient is in A. If P is nonnegative, it is the remainder. Otherwise, it needs to be restored (i.e., add b), and then it will be the remainder. A numerical example is given in Figure A.3(b). Note that the sign of P must be tested before shifting, since the sign bit can be lost when shifting. However, because of two's complement arithmetic (discussed in the next section), the net result of shifting followed by the appropriate add/subtract operation will be the correct value. This comes about because the result of each step is a number r with $|r| \leq b$.

If a and b are unsigned numbers in the range $0 \leq a,b \leq 2^n - 1$, then the multiplier in Figure A.2 will work if register P is n bits long. However, for division, P must be extended to $n + 1$ bits in order to detect the sign of P. Thus the adder must also have $n + 1$ bits.

Why would anyone implement restoring division, which uses the same hardware as nonrestoring division (the control is slightly different) but involves an extra addition? In fact, the usual implementation for restoring division doesn't literally perform an add in step 4. Rather, the sign resulting from the subtraction is tested, and only if the sum is nonnegative is it loaded back into the P register.

As a final point, before beginning to divide, the hardware must check to see if the divisor is zero.

P	A	
00000	1110	Divide 14 = 1110 by 3 = 11. B always contains 0011
00001	110	step (1): shift
−00011		step (2): subtract
−00010	110**0**	step (3): result is negative, set quotient bit to 0
00001	110**0**	step (4): restore
00011	10**0**	step (1): shift
−00011		step (2): subtract
00000	10**01**	step (3): result is nonnegative, set quotient bit to 1
00001	0**01**	step (1): shift
−00011		step (2): subtract
−00010	0**010**	step (3): result is negative, set quotient bit 0
00001	0**010**	step (4): restore
00010	**010**	step (1): shift
−00011		step (2): subtract
−00001	**0100**	step (3): result is negative, set quotient bit to 0
00010	**0100**	step (4): restore. The quotient is 0100 and the remainder is 00010.

(a)

00000	1110	Divide 14 = 1110 by 3 = 11. B always contains 0011
00001	110	step (1b): shift
+11101		step (2b): subtract *b* (add 2's complement)
11110	110**0**	step (3): P is negative, so set quotient bit to 0
11101	10**0**	step (1a): shift
+00011		step (2a): add *b*
00000	10**01**	step (3): P is nonnegative, so set quotient bit to 1
00001	0**01**	step (1b): shift
+11101		step (2b): subtract *b*
11110	0**010**	step (3): P is negative, so set quotient bit to 0
11100	**010**	step (1a): shift
+00011		step (2a): add *b*
11111	**0100**	step (3): P is negative, so set quotient bit to 0
+00011		remainder is negative, so do final restore step
00010		The quotient is 0100 and the remainder is 00010

(b)

FIGURE A.3 Numerical example of (a) restoring division and (b) nonrestoring division.

Signed Numbers

There are four methods commonly used to represent signed n-bit numbers: *sign magnitude, two's complement, one's complement,* and *biased.* In the sign-magnitude system, the high-order bit is the sign bit, and the low-order $n-1$ bits are the magnitude of the number. In the two's complement system, a number and its negative add up to 2^n. In one's complement, the negative of a number is obtained by complementing each bit. In a biased system, a fixed bias is picked so that the sum of the bias and the number being represented will always be nonnegative. A number is represented by first adding it to the bias, and then encoding the sum as an ordinary unsigned number.

Example: How is –3 expressed in each of these formats?

Answer: The binary representation of 3 is 0011_2. In signed magnitude, –0011 = 1011. In two's complement $0011_2 + 1101_2 = 8$, so –0011 = 1101. In one's complement, –0011 = 1100. Using a bias of 8, 3 is represented by 1011, and –3 by 0101.

The most widely used system for representing integers, two's complement, is the system we will use here; one's complement is discussed in the Exercises. One reason for the popularity of two's complement is that addition is extremely simple: Simply discard the carry out from the high-order bit. To add 5 + –2, for example, add 0101 and 1110 to obtain 0011, resulting in the correct value of 3. A useful formula for the value of a two's complement number $a_{n-1}a_{n-2} \cdots a_1 a_0$ is

A.2.3 $$-a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + \cdots + a_1 2^1 + a_0$$

Overflow occurs when the result of the operation does not fit in the representation being used. For example, if unsigned numbers are being represented using four bits, then $6 = 0110_2$, and $11 = 1011_2$. Their sum (17) overflows because its binary equivalent (10001_2) doesn't fit into four bits. For unsigned numbers, detecting overflow is easy; it occurs exactly when there is a carry out of the most significant bit. For two's complement, things are trickier: Overflow occurs exactly when the carry into the high-order bit is different from the (to be discarded) carry out of the high-order bit. In the example of 5 + –2 above, a 1 is carried both into and out of the leftmost bit, avoiding overflow.

Negating a two's complement number involves complementing each bit and then adding 1. For instance, to negate 0011, complement it to get 1100 and then add 1 to get 1101. Thus, to implement $a - b$ using an adder, simply feed a and $\overline{b}$ (where $\overline{b}$ is the number obtained by complementing each bit of b) into the adder, and set the low-order, carry-in bit to 1. This explains why the rightmost adder in Figure A.1 is a full adder.

Multiplying two's complement numbers is not quite as simple as adding them. The obvious approach is to convert both operands to be nonnegative, do an unsigned multiplication, and then (if the original operands were of opposite signs) negate the result. Although this is conceptually simple, it requires extra time and hardware. Here is a better approach: Suppose that we are multiplying *a* times *b* using the hardware shown in Figure A.2(a) (page A-4). Register A is loaded with the number *a*; B is loaded with *b*. Since the contents of register B is always *b*, we will use B and *b* interchangeably. The first thing to do when multiplying two's complement numbers is to ensure that when P is shifted, it is shifted arithmetically; that is, the bit shifted into the high-order bit of P should be the sign bit of P. Note that our *n*-bit–wide adder will now be adding *n*-bit two's complement numbers between -2^{n-1} and $2^{n-1} - 1$.

Next, suppose *a* is negative. The method for handling this case is called *Booth recoding*. Booth recoding is a very basic technique in computer arithmetic and will play a key role in Section A.9. Observe that multiplying by 0111_2 is the same as multiplying by $1000_2 - 1$. To perform this multiplication, subtract *b* from register P in the first multiplication cycle. Add zero in the second and third cycles. In the fourth cycle, add *b*. To apply this technique to a negative multiplier like $-4 = 1100_2$, think of it as an unsigned number and write it as $10000_2 - 0100_2$. If the multiplication algorithm only involves *n* steps ($n = 4$ in this case), the 10000_2 term is ignored, and we end up subtracting $0100_2 = 4$ times the multiplier—exactly the right answer. The advantage of Booth recoding is that it works equally well for positive and negative multipliers. To deal with negative values of *a*, then, all that is required is to sometimes subtract *b* from P, instead of either adding *b* or 0 to P. Here are the precise rules: If the initial content of A is $a_{n-1} \cdots a_0$, then at the *i*th multiply step, the low-order bit of register A is a_i, and

1. If $a_i = 0$ and $a_{i-1} = 0$ then add 0.
2. If $a_i = 0$ and $a_{i-1} = 1$ then add B.
3. If $a_i = 1$ and $a_{i-1} = 0$ then subtract B.
4. If $a_i = 1$ and $a_{i-1} = 1$ then add 0.

For the first step, when $i = 0$, take a_{i-1} to be 0.

Example:

When multiplying –6 times –5, what is the sequence of values in the (P,A) register pair?

Answer:

Initially, P is zero and A holds $-6 = 1010_2$. From Figure A.4, in the first step 0 is added to P giving (P,A) = 0000 1010. After shifting (P,A) = 0000 0101. In the next step, Figure A.4 shows that 0101 is added to P giving (P,A) = 0101 0101. Continuing, (P,A) = 0010 1010, 1101 1010, 1110 1101, 0011 1101, and finally 0001 1110.

The four cases above can be restated as saying that in the *i*th step you should add $(a_{i-1} - a_i)$B to P. With this observation, it is easy to verify that these rules work, because the result of all the additions is

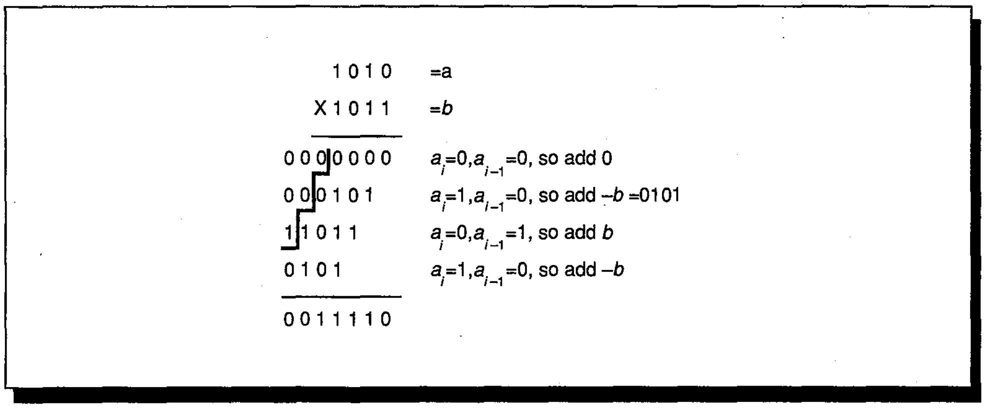
$$\sum_{i=0}^{n-1} b(a_{i-1} - a_i)2^i = b(-a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + \cdots + a_1 2 + a_0)$$

From Equation A.2.3 (page A-7), the quantity in parenthesis is the value of A as a two's complement number.

The simplest way to implement the rules for Booth recoding is to extend the A register one bit to the right so that this new bit will contain a_{i-1}. Unlike the naive method of inverting any negative operands, this technique doesn't require extra steps or any special casing for negative operands. It has only a slightly more complicated control logic. If the multiplier is being shared with a divider, there will already be the capability for subtracting *b*, rather than adding it. To summarize, a simple method for handling two's complement multiplication is to pay attention to the sign of P when shifting it right, and to save the most recently shifted off bit of A to use in deciding whether to add or subtract *b* from P.

The reason for the term "recoding" is as follows. Consider representing numbers using 1, 0, and $\bar{1}$ where $\bar{1}$ represents −1; as an example, this allows us to also represent (recode) 0111 as $100\bar{1}$. Imagine a multiplication algorithm that worked as follows: Put a recoded number into the A register. If the low-order bit of A is 1, then add B. If it is $\bar{1}$, then subtract B. If the low-order bit is 0, then add 0. This imaginary algorithm has exactly the same effect as the Booth recoding method given above.

Booth recoding is usually the best method for designing hardware that operates on signed numbers. For hardware that doesn't directly implement it, however, performing Booth recoding in software or microcode is usually too slow, due to the conditional tests and branches. If the hardware supports arithmetic shifts (so that negative *b* is handled correctly), then the following



```
     1 0 1 0    =a
    X1 0 1 1    =b
 0 0 0|0 0 0 0  a_i=0,a_i-1=0, so add 0
 0 0|0 1 0 1    a_i=1,a_i-1=0, so add -b =0101
 1|1 0 1 1      a_i=0,a_i-1=1, so add b
 0 1 0 1        a_i=1,a_i-1=0, so add -b
 0 0 1 1 1 1 0
```

FIGURE A.4. Multiplication of *a* = −6 by *b* = −5 to get 30 using Booth recoding. The digits to the left of the jagged line are the sign-extended digits.

method can be used. Treat the multiplier a as if it were an unsigned number, and perform $n-1$ multiply steps. If $a < 0$ (in which case there will be a 1 in the low-order bit of the A register at this point), then subtract b from P; otherwise ($a \geq 0$) neither add nor subtract. In either case, do a final shift (for a total of n shifts) to get the low-order bit of the product into the low-order position of A. This works because it amounts to multiplying b by $-a_{n-1}2^{n-1} + \cdots + a_1 2 + a_0$, which is the value of $a_{n-1} \cdots a_0$ as a two's complement number by Equation A.2.3. If the hardware doesn't support arithmetic shift, then converting the operands to be nonnegative is probably the best approach.

Two final remarks: A good way to test a signed-multiply routine is to try $-2^{n-1} \times -2^{n-1}$, since this is the only case that produces a $2n-1$ bit result. Unlike multiplication, division is usually performed in hardware by converting the operands to be nonnegative and then doing an unsigned divide; because division is substantially slower (and less frequent) than multiplication, the extra time used to manipulate the signs has less impact than it does on multiplication.

Systems Issues

When designing an instruction set, there are a number of issues related to integer arithmetic that need to be resolved. Several of them are discussed here.

First, what should be done about integer overflow? This situation is complicated by the fact that detecting overflow is different depending on whether the operands are signed or unsigned integers. Consider signed arithmetic first. There are three approaches: Set a bit on overflow, trap on overflow, or do nothing on overflow. In the last case, software has to check whether or not an overflow occurred. The most convenient solution for the programmer is to have an enable bit. If this bit is turned on, then overflow causes a trap. If it is turned off, then overflow sets a bit. The advantage of this approach is that both trapping and nontrapping operations require only one instruction. Furthermore, as we will see in Section A.7, this is analogous to how the IEEE floating-point standard handles floating-point overflow. Figure A.5 shows how some common machines treat overflow.

What about unsigned addition? Notice that none of the architectures in Figure A.5 trap on unsigned overflow. The reason for this is that the primary use of unsigned arithmetic is in manipulating addresses. It is convenient to be able to subtract from an unsigned address by adding. For example, when $n = 4$, we can subtract 2 from the unsigned address $10 = 1010_2$ by adding $14 = 1110_2$. Even though $1010_2 + 1110_2$ sums to the answer we wanted ($1000_2 = 8$), this operation has an unsigned overflow. In other words, addresses are treated as both signed and unsigned numbers, making an overflow trap useless for address calculations.

A second issue concerns multiplication. Should the result of multiplying two n-bit numbers be a $2n$-bit result, or should multiplication just return the low-order n bits, signaling overflow if the result doesn't fit in n bits? The argument in favor of an n-bit result is that in virtually all high-level languages, multiplication is an operation whose arguments are integer variables and whose result is an integer variable of the same type. Therefore, there is no way to generate code that utilizes a double-precision result. The argument in favor of a 2*n*-bit result is that it can be used by an assembly language routine to speed up multiplication of multiple-precision integers substantially (by about a factor of 3).

A third issue concerns machines that want to execute one instruction every cycle. It is rarely practical to perform a multiplication or division in the same amount of time that an addition or register–register move takes. There are three possible approaches to this problem. The first is to have a single-cycle *multiply-step* instruction. This might do one step of the Booth algorithm. The second approach is to do integer multiplication in the floating-point unit and have it be part of the floating-point instruction set. (This is what DLX does.) The third approach is to have an autonomous unit in the CPU do the multiplication. In this case, the result can either be guaranteed to be delivered in a fixed number of cycles—and the compiler charged with waiting the proper amount of time—or there can be an interlock. The same comments apply to division as well. As examples, the SPARC has a multiply-step instruction but no divide-step instruction, and the MIPS R3000 has an autonomous unit that does multiplication and division (see Section E-6 for new extensions to SPARC for arithmetic). The designers of the HP Precision Architecture did an especially thorough job of analyzing the frequency of the operands for multiplication and division, and based their multiply and divide steps accordingly. (See Magenheimer et al. [1988] for details.)

A potential pitfall worth mentioning concerns multiple-precision addition. Many instruction sets offer a variant of the ADD instruction that adds three operands: two *n*-bit numbers together with a third single-bit number. This third number is the carry from the previous addition. Since the multiple-precision number will typically be stored in an array, it is important to be able to increment the array pointer without destroying the carry bit.

Machine	Trap on signed overflow?	Trap on unsigned overflow?	Set bit on signed overflow?	Set bit on unsigned overflow?
VAX	If enable is on	No	Yes. ADD sets V bit.	Yes. ADD sets C bit.
IBM 370	If enable is on	No	Yes. ADD sets cond code.	Yes. Logical ADD sets cond code.
Intel 8086	No	No	Yes. ADD sets V bit.	Yes. ADD sets C bit.
MIPS R3000	There are 2 ADD instructions: one always traps, the other never does.	No	No. Software must deduce it from sign of operands and result.	
SPARC	No	No	ADDCC sets V bit. ADD does not.	ADDCC sets C bit. ADD does not.

FIGURE A.5 Summary of how various machines handle integer overflow. Both the 8086 and SPARC have an instruction that traps if the V bit is set, so the cost of trapping on overflow is one extra instruction.

A.3 Floating Point

Introduction

Many applications require numbers that aren't integers. There are a number of ways that nonintegers can be represented. One is to use *fixed point*; that is, use integer arithmetic and simply imagine the binary point somewhere other than just to the right of the least significant digit. Adding two such numbers can be done with an integer add, whereas multiplication requires some extra shifting. Other representations that have been proposed involve storing the logarithm of a number and doing multiplication by adding the logarithms, or using a pair of integers (a,b) to represent the fraction a/b. However, there is only one noninteger representation that has gained widespread use, and that is the floating-point representation. In this system, a computer word is divided into two parts, an exponent and a significand. As an example, an exponent of –2 and significand of 1.5 might represent the number $1.5 \times 2^{-2} = 0.375$. The advantages of standardizing a particular representation are obvious. Numerical analysts can build up high-quality software libraries, computer designers can develop techniques for implementing high-performance hardware, and hardware vendors can build standard accelerators. Given the predominance of the floating-point representation, it appears unlikely that any other representation will come into widespread use.

A key fact about floating-point instructions is that their semantics are not as clear cut as the semantics of the rest of the instruction set, and in the past the behavior of floating-point operations varied considerably from one computer family to the next. The variations involved such things as the number of bits allocated to the exponent and significand, the range of exponents, how rounding was carried out, and the actions taken on exceptional conditions like underflow and overflow. Computer architecture books used to dispense advice on how to deal with all these details, but fortunately this is no longer necessary. That's because the computer industry is rapidly converging on the format specified by IEEE standard 754-1985. The advantages of using a standard variant of floating point are similar to those for using floating point over other noninteger representations. In this chapter we will discuss only the IEEE version of floating point. For further reading see IEEE [1985], Cody et al. [1984], Cody [1988], and Goldberg [1989].

Overview of the IEEE Standard

Probably the most notable feature of the standard is that it requires computation to continue in the face of exceptional conditions, such as dividing by zero or taking the square root of a negative number. The result of taking the square root of a negative number is a *NaN* (*N*ot *a* *N*umber), a bit pattern that does not represent an ordinary number. As an example of how NaNs might be useful, consider the code for a zero finder that takes a function F as an argument and evaluates F at various points to determine a zero for it. If the zero finder accidentally probes outside the valid values for F, F may well cause an exception. Writing a zero finder that deals with this case is highly language and operating-system dependent, because it relies on how the operating system reacts to exceptions and how this reaction is mapped back into the programming language. In IEEE arithmetic it is easy to write a zero finder that handles this situation and runs on many different systems. After each evaluation of F, it simply checks to see if F has returned a NaN; if so, it knows it has probed outside the domain of F.

Because of the rules for performing arithmetic with NaNs, writing floating-point subroutines that can accept NaN as an argument rarely requires any special case checks. Suppose that arccos is computed in terms of arctan, using the formula $\arccos x = 2 \arctan(\sqrt{(1-x)/(1+x)})$. If arctan handles an argument of NaN properly, arccos will automatically do so too. That's because the IEEE standard specifies that when an argument of an operation is a NaN, the result should be a NaN. Therefore if x is a NaN, $1+x$, $1-x$, $(1+x)/(1-x)$ and $\sqrt{(1-x)/(1+x)}$ will also be NaNs. No checking for NaNs is required.

While the result of $\sqrt{-1}$ is a NaN, the result of 1/0 is not a NaN, but $+\infty$, which is another special value. The standard defines arithmetic on infinities (including $-\infty$) using rules such as $1/\infty = 0$. The formula $\arccos x = 2 \arctan(\sqrt{(1-x)/(1+x)})$ illustrates how infinity arithmetic can be used. Since $\arctan x$ asymptotically approaches $\pi/2$ as x approaches ∞, it is natural to define $\arctan(\infty) = \pi/2$, in which case $\arccos(-1)$ will automatically be computed correctly as $2 \arctan(\infty) = \pi$.

Another feature of the IEEE standard with implications for hardware is the rounding rule. When operating on two floating-point numbers, the result is usually a number that cannot be exactly represented as another floating-point number. For example, in a floating-point system using base 10 and two significant digits, $2.1 \times 0.5 = 1.05$. This needs to be rounded to two digits. Should it be rounded to 1.0 or 1.1? In the IEEE standard, such halfway cases are rounded to the number whose low-order digit is even. That is, 1.05 rounds to 1.0, not 1.1. The standard actually has four *rounding modes*. The default is *round to nearest*, which rounds to an even number in the case of ties. The other modes are round toward 0, round toward $+\infty$ and round toward $-\infty$.

The standard specifies four precisions: *single*, *single extended*, *double*, and *double extended*. The properties of these precisions are summarized in Figure A.6 (page A-14). Implementations are not required to have all four precisions, but are encouraged to support either the combination of single and single extended or all of single, double, and double extended. Let us consider single precision in more detail. Single-precision numbers are represented using 32 bits: 1 for the sign, 8 for the exponent, and 23 for the fraction. The exponent is a signed number represented using the bias method (as explained in Section A.2 above) with a bias of 127. We will always use the term *exponent field* to mean the unsigned number contained in bits one through nine and *exponent* to mean the power to which two is to be raised. (In the standard these are called the "biased exponent" and the "unbiased exponent," respectively.) The fraction represents a number less than one, but the *significand* of the floating-point number is one plus the fraction part. In other words, if e is the value of the exponent field and f is the value of the fraction field, the number being represented is $1.f \times 2^{e-127}$.

Example: What single-precision number does the following 32-bit word represent?

```
1 10000001 01000000000000000000000
```

Answer: Considered as an unsigned number, the exponent field is 129, making the value of the exponent 129 − 127 = 2. The fraction part is $.01_2 = .25$, making the significand 1.25. Thus, this bit pattern represents the number $-1.25 \times 2^2 = -5$.

The fractional part of a floating-point number (.25 in the example above) must not be confused with the significand, which is one plus the fractional part. The leading 1 in the significand $1.f$ does not appear in the representation; that is, the leading bit is implicit. When performing arithmetic on IEEE format numbers, the fraction part normally needs to be *unpacked,* which is to say the implicit one needs to be made explicit.

In Figure A.6, the range of exponents for single precision is −126 to 127; accordingly, the exponent field ranges from 1 to 254. The exponent fields of 0 and 255 are used to represent special values. When the exponent field is 255, a zero fraction field represents infinity, and a nonzero fraction field represents a NaN. Thus, there is an entire family of NaNs. When the exponent and fraction fields are zero, then the number represented is zero. Because ordinary numbers always have a significand greater than or equal to 1—and are thus never zero—a special convention such as this is required to represent zero.

A zero exponent field and nonzero fraction part represent a *denormal* number, also sometimes called a *subnormal* number. These numbers make up the most controversial part of the standard. Later, in the discussion of multiplication, we will see why they are difficult to implement in hardware. In many floating-point systems if E_{min} is the smallest exponent, a number less than $1.0 \times 2^{E_{\text{min}}}$

	Single	Single extended	Double	Double extended
p (bits of precision)	24	≥ 32	53	≥ 64
E_{max}	127	≥ 1023	1023	≥ 16383
E_{min}	−126	≤ -1022	−1022	≤ -16382
Exponent bias	127		1023	

FIGURE A.6 Format parameters for the IEEE 754 floating-point standard. The first row gives the number of bits in the significand. The blank boxes are unspecified parameters.

cannot be represented, and a floating-point operation that results in a number less than this is simply flushed to zero. In the IEEE standard, on the other hand, numbers less than $1.0 \times 2^{E_{\min}}$ are represented by shifting their fraction part to the right. This is called *gradual underflow*. Thus, as numbers decrease in magnitude below $2^{E_{\min}}$, they gradually lose their significance and are only represented by zero when all their significance has been shifted out. For example, in base 10 with 4 significant figures, let $x = 1.234 \times 2^{E_{\min}}$. Then $x/10 = 0.123 \times 10^{E_{\min}}$, having lost a digit of precision; $x/100$ and $x/1000$ have even less precision, while $x/10000$ is finally small enough to be rounded to zero. Denormalized numbers are implemented by having a word with a zero exponent field represent the number $0.f \times 2^{E_{\min}}$. One of the advantages of gradual underflow is that when it is used, if $x \neq y$, then $x - y \neq 0$. In a flush-to-zero system, this is not always true.

The primary reason why the IEEE standard, like most other floating-point formats, uses biased exponents is that it means nonnegative numbers are ordered in the same way as integers. That is, the magnitude of floating-point numbers can be compared using an integer comparator. Another (related) advantage is that zero is represented by a word of all zeros. The down side of biased exponents is that adding them is slightly awkward, because it requires that the bias be subtracted from their sum.

As the IEEE standard becomes more widespread, it will become easier to port software and to write portable libraries that deal with floating-point exceptions. But the standard also has some drawbacks:

1. It was originally intended for microprocessors, so the requirements of high-performance implementations were not given high priority.

2. The standard contains optional parts. This results in difficult decisions for implementors—which parts should they implement?—and for portable software writers—should they avoid using any of the optional parts of the standard?

3. Gradual underflow has usually been implemented in a way that is orders of magnitude slower than flush to zero, so users often disable it.

4. There is as yet no industrial-strength, public-domain, IEEE floating-point test suite.

Although the standard may ultimately improve the quality of floating-point libraries, this has yet to happen because of the large base of VAXes, IBM/370s, and Crays, as well as the fact that there is no corresponding standard for how to access its features in software. On the other hand, both DEC and IBM have recently introduced machines that use IEEE arithmetic.

Some final comments on the standard:

1. Unlike most standards, IEEE 754 did not ratify or refine any existing system. Although most of the features of the standard appeared in at least one previous computer system, it is substantially different from what was current practice at the time.

2. The standard says nothing about integer arithmetic or about transcendental functions (sin, cos, exp, and so forth). In particular, it says nothing about the accuracy that transcendentals should have, and it says nothing about the exceptional values of transcendentals, such as 0^0.

3. It is intended that a computer **system**—that is, some combination of hardware and software—will implement the standard. Thus, there is nothing wrong with designing hardware that does not completely implement the standard, as long as there is some way for software to provide what the hardware does not. In fact, the best design may well involve having rare cases handled by software.

A.4 Floating-Point Addition

There are two differences between floating-point arithmetic and integer arithmetic: An exponent field must be manipulated, in addition to the fraction field, and the result of a floating-point operation usually has to be rounded in order to be represented by another floating-point number of the same precision.

Rounding

The IEEE standard specifies that the result of an arithmetic operation should be the same as it would be if computed exactly and then rounded using the current rounding mode. The most difficult mode to implement is the default mode—round toward the nearest value (and round halfway cases to even). The naive approach to complying with the IEEE standard is to compute the sum exactly and then round. This would be quite expensive, since it would require a very long adder. To see how to satisfy the standard with less hardware, we will consider some examples.

There are two ways that rounding can occur during addition. For purposes of illustration we will use base 10, which is more natural for humans, and three significant digits. The first case requires rounding due to carry out on the left, as illustrated in Figure A.7(a). The second case requires rounding due to unequal exponents, as in Figure A.7(b). Figure A.7(c) shows that it is possible for both situations to occur simultaneously. In each of these cases, the sum must be computed to more than three places in order to perform rounding. In one case—when subtracting nearby numbers, as in Figure A.7(d)—the sum must be computed to more than three places, even though no rounding occurs. By temporarily ignoring the round-to-even requirement, each of these examples can be implemented with a four-digit-wide adder (that is, using one additional digit). Thus, in Figure A.7(b) the rightmost 6 of 2.56 can simply be dropped before adding. But there is one case, shown in Figure A.7(e), in which four digits are not enough. If the low-order digit of .0376 were shifted off, the answer would have been .973 instead of .972. However, it is easy to check (disregarding round to even) that two extra digits are always enough. These extra digits are called the *guard* and *round* digits.

The round-to-even rule introduces an extra complication. Figure A.7(f) shows an example with five significant digits. It might appear at first that one needs to keep double the number of digits to perform round to even, as the rightmost 1 in 2.5001 determines whether the result will be 4.5676 or 4.5677.

Upon a little reflection one can see that it is only necessary to know whether or not there are any nonzero digits past the guard and round positions. This information can be stored in a single bit, usually called the *sticky bit*, which is implemented by examining each digit as it is shifted off. As soon as a nonzero digit appears, the sticky bit is set on and remains stuck on. To implement round to even, simply append the sticky bit to the right of the round digit just before rounding.

$$\begin{array}{lrl} & 2.34 \times 10^{2} & \\ \text{a)} & +8.51 \times 10^{2} & \\ \hline & 10.85 \times 10^{2} & \text{rounds to } 1.08 \times 10^{3} \end{array}$$

$$\begin{array}{lrrl} & 2.34 \times 10^{2} & 2.34 \times 10^{2} & \\ \text{b)} & +2.56 \times 10^{0} & +.0256 \times 10^{2} & \\ \cline{3-3} & & 2.36\underset{g}{5}\underset{r}{6} \times 10^{2} & \text{rounds to } 2.37 \times 10^{2} \end{array}$$

$$\begin{array}{lrl} & 9.51 \times 10^{2} & \\ \text{c)} & +.642 \times 10^{2} & \\ \hline & 10.15\underset{g}{2} \times 10^{2} & \text{rounds to } 1.02 \times 10^{3} \end{array}$$

$$\begin{array}{lr} & 1.47 \times 10^{2} \\ \text{d)} & -.876 \times 10^{2} \\ \hline & .59\underset{g}{4} \times 10^{2} \end{array}$$

$$\begin{array}{lrl} & 1.01 \times 10^{2} & \\ \text{e)} & -.0376 \times 10^{2} & \\ \hline & .97\underset{g}{2}\underset{r}{4} \times 10^{2} & \text{rounds to } .972 \times 10^{2} \end{array}$$

$$\begin{array}{lrrl} & 4.5674 \times 10^{0} & 4.5674 & \\ \text{f)} & 2.5001 \times 10^{-4} & +\ .00025001 & \\ \cline{3-3} & & 4.5676\underset{g}{5}\underset{r}{0}01 & \text{rounds to } 4.5677 \end{array}$$

FIGURE A.7 Examples of rounding. In (a) there is rounding because of carry out on the left and in (b) because of unequal exponents, whereas in (c) both occur. Example (d) shows that one extra place must be kept even if there is no rounding, while (e) shows the situation in which two extra digits are needed. Finally (f), where $p = 5$, illustrates why a sticky bit is necessary to perform round to even. The letters g and r are placed under the guard and round digits.

The Addition Algorithm

The notations e_i and s_i are used here for the exponent and significand fields of the floating-point number a_i. This means that the floating-point number has been unpacked and that s_i has an explicit leading bit. The basic procedure for adding two floating-point numbers a_1 and a_2 is straightforward and involves five steps.

1. If $e_1 < e_2$, swap the operands so that the difference of the exponents satisfies $d = e_1 - e_2 \geq 0$. Tentatively set the exponent of the result to e_1.

2. Shift s_2 by $d = e_1 - e_2$ places to the right. More precisely, put s_2 into a p-bit register and then extend that register MIN(2,d) bits to the right. Shift s_2 d places to the right. If $d > 2$, set the sticky bit to the logical OR of the $d - 2$ bits that are shifted out of the extended register. Of the two extended bits, the most significant is the guard bit; the least significant is the round bit.

3. Append the sticky bit to s_2, and then add the two signed-magnitude fraction fields in a $p + 3$ bit adder. Call this preliminary sum S.

4. If there was a carry out from the most significant place in the previous step, shift the magnitude of S right by one. Otherwise, shift it left until it is normalized. Adjust the exponent of the result accordingly. The round bit is now set to the (p + 1)-st bit of the magnitude of S, and the sticky bit to the logical OR of all the bits to the right of the round bit.

5. Round the result using Figure A.8. If a table entry is nonempty, add 1 to the magnitude of S. Thus, if $S \geq 0$, you will be computing $S + 1$, otherwise $S - 1$.

The guard and round bits before shifting are marked in each of the examples of Figure A.7 (page A-17).

Example: Show how the addition algorithm proceeds on the operands of Figure A.7(f) when round to nearest is in effect.

Answer: In step 1, $e_1 = 0 > e_2 = -3$, so $d = 3$ and no swapping is necessary. In step 2, $g = 5$, $r = 0$, and sticky is the OR of 0, 0, and 1; hence, sticky is 1. In step 3 the numbers to be added are 4.5674 and 0.0002501, so the preliminary sum is $S = 4.5676501$. In step 4 there is no carry out, so d is still 3. The round bit is 5, and the sticky bit is $1 = 0 \vee 1$. In step 5, consulting the table tells us that because round and sticky are both nonzero, we must add 1 to the fifth digit of S, changing S from 45676 to 45676 + 1 = 45677.

Step 3 involves adding sign-magnitude numbers, and itself has three steps:

3a. Convert any negative numbers to two's complement.

3b. Perform a (p + 4)-bit two's complement addition (p + 3 bits of magnitude, 1 bit for the sign).

3c. If the result is negative, perform another two's complementation to put the result back into sign-magnitude form.

As is apparent from this, addition is quite a complicated operation. Here is one trick that can speed it up. A pair of numbers will only need to be variably shifted once, in either step 2 or step 4, but not in both. The reason is simple: If $|e_1 - e_2| > 1$, then step 4 can require a shift of at most one place. And if $|e_1 - e_2| \leq 1$, then step 2 obviously requires a shift of at most one step. A nonpipelined adder can exploit this and reduce the number of steps from five to four. An adder that uses each of the above steps as a pipeline stage can also use this reduction, though it requires duplicating the shifter and adder.

Step 3 can be time consuming, because it can involve as many as four additions: two to negate both operands (two's complementation done by performing a bitwise complementation followed by adding 1), a third for the addition itself, and then a fourth to negate the result. There are a number of ways to speed up this step. We have already seen that 1 can be added to a sum essentially for free by setting the low-order, carry-in bit of the adder to 1. If both operands are negative, we can set their sign bits to zero, remembering to negate the result. The add required when negating the result can be combined with the rounding step (which must be prepared to do an add anyway).

The rounding step requires a second full-precision add in addition to the one in step 3. It is possible to combine these into a single add. Observe that at the end of step 2, the *g, r,* and *s* bits are known; thus it is also known whether or not to round up, adding 1 to the *p*th most significant bit. What is not known is the position of the *p*th most significant bit, since its location depends on the result of the add in step 3; when adding numbers of the same sign, that position is determined by whether there is a carry out of the most significant bit. Therefore, the way to eliminate step 5 is to add in the round-up bit (if necessary) as part of step 3. Because the position is unknown, two versions of step 3 must be performed using two adders in parallel. Each adder assumes one of the two possibilities for the position where the round-up bit goes. This technique for reducing the number of addition steps is used on the Intel 860 [Kohn 1989]. When rounding, there is one complication that can arise: The addition of 1 could cause a carry out of the high-order bit. This case occurs only when the value of *S* is 11···11.

Rounding mode	$S \geq 0$	$S < 0$
$-\infty$		+1 if $r \vee s$
$+\infty$	+1 if $r \vee s$	
0		
Nearest	+1 if $r \wedge \bar{s} \wedge p_0$ or $r \wedge s$	+1 if $r \wedge \bar{s} \wedge p_0$ or $r \wedge s$

FIGURE A.8 Rules for implementing the IEEE rounding modes. Blank boxes mean that the *p* most significant bits of the preliminary sum *S* are the actual sum bits. If the condition in the box is true, add 1 to the *p*th most significant bit of *S*. The symbols *r* and *s* represent the round and sticky bits, while p_0 is the *p*th most significant bit of *S*.

Denormalized Numbers

Very little changes in the above description if one of the inputs is a denormal number. There must be a test to see if the exponent field is 0. If it is, then when unpacking the significand there will not be a leading 1. By setting the exponent field to 1 when unpacking a denormal, the shifting rules in steps 1–5 are still correct.

In order to deal with denormalized outputs, step 4 must be modified slightly. The value in the P register is shifted left until P is normalized, or until the exponent becomes E_{min} (that is, the exponent field becomes 1). If the exponent is E_{min}, and if after rounding, the high-order bit of P is 1, then the result is a normalized number and should be packed in the usual way, by omitting the 1. If, on the other hand, the high-order bit is 0, the result is denormal, and when the result is unpacked the exponent field must be set to 0.

Incidentally, detecting overflow is very easy. It can only happen if step 4 involves a shift right, and if the exponent field at that point is bumped up to 255 in single precision (or 2047 for double precision), or if this occurs after rounding.

Detecting underflow is complicated by the fact that it depends on whether there is a user trap handler. The IEEE standard specifies that if user trap handlers are enabled, the system must trap if the result is denormal. On the other hand, if trap handlers are disabled, then the underflow flag is set only if there is a loss of accuracy—that is, if the result must be rounded. The rationale for this is that if no accuracy is lost on an underflow, there is no point in setting a warning flag. But if a trap handler is enabled, the user might be trying to simulate flush-to-zero and should therefore be notified whenever a result dips below $1.0 \times 2^{E_{\mathrm{min}}}$. This discussion is relevant for addition in that an addition or subtraction resulting in a denormal number will always be exact; because no accuracy can be lost to underflow, there is no need to set the underflow flag.

A.5 Floating-Point Multiplication

Floating-point multiplication is much like integer multiplication. Because floating-point numbers are stored in sign-magnitude form, the multiplier need only deal with unsigned numbers (although we have seen that Booth recoding handles signed two's complement numbers painlessly). If the fractions are unsigned p-bit numbers, then the product can have as many as $2p$ bits and must be rounded to a p-bit number. Besides multiplying the fraction parts, the exponent fields must be added, and the bias then subtracted from their sum.

Here is a straightforward method of handling rounding using the multiplier of Figure A.2 (page A-4): Multiply the two fractions to obtain a $2p$-bit product in the (P,A) registers. During the multiplication, the first $p - 2$ times a bit is shifted into the A register, OR it into the sticky bit. After the end of all the multiply steps, the high-order bit of A is the guard bit, and the second high-order bit is the round bit. There are two cases:

1. The high-order bit of P is 0. Shift P left 1 bit, shifting in the g bit from A. Shifting the rest of A is not necessary.
2. The high-order bit of P is 1. Set $s := s \vee r$ and $r := g$, and add 1 to the exponent.

Now use the rules in Figure A.8 (page A-19) to round the result, adding the 1 (if necessary) into the low-order bit of P. The fraction (in unpacked form) is in the P register. Recall that the rounding operation can cause a carry out of the most significant bit. A good discussion of more efficient ways to implement rounding is in Santoro, Bewick, and Horowitz [1989].

Detecting overflow and underflow is slightly tricky. Consider the case of single precision. The exponent fields must be added together with –127. If the addition is done in a 10-bit adder, $-127 = 1110000001_2$, and overflow occurs when the high-order bits of the sum are 01 or if the sum is 0011111111. Underflow occurs when the high-order bits are 11 or the sum is 0000000000. Alternatively, the addition can be done using only an 8-bit adder. Simply add both exponents and $-127 = 10000001_2$. If the high-order bits of the exponent fields are different, no over/underflow is possible. If the high-order bits are both 1, the result has overflowed if it has 0 in the high-order bit or if it is 1111111. If both the exponents have high-order bits of zero, underflow has occurred if the sum has a high-order bit of 1, or if the sum is 00000000.

Denormals

From the description of the multiplication algorithm, one can see that after doing an integer multiplication on the fractions, the final result is obtained with at most one shift. With denormals, the situation changes completely. Suppose the input is normalized, but the output is denormal, so that in single precision the product has an exponent e with $e < -126$. Then the result must be shifted right by $-e - 126$ places. This requires extra hardware (a barrel shifter that wouldn't otherwise be needed) and extra time. The situation with denormal inputs isn't any better, because even if the final result is a normalized number, a variable shift is still required. Thus, high-performance, floating-point multipliers often do not handle denormalized numbers, but instead trap, letting software handle them. There are a few practical codes that generate many underflows, even when working properly, and these programs usually run quite a bit slower on systems that require denormals to be processed by a trap handler.

One procedure followed by some floating-point units is to have the multiplier deliver denormalized outputs in *wrapped* form. That is, the fraction part is normalized, and the exponent is wrapped around to a large positive number. This is exactly the result when following the multiplication algorithm for normalized numbers given above. Since the addition unit must have a barrel shifter, it is usually straightforward to provide a way to convert wrapped numbers into their correct denormalized form by passing them through the adder. However, if a trap handler has to intervene in order to send wrapped numbers into the adder, multiplication will still be slowed down substantially.

There are some fine points that occur when a multiplication results in a denormal number. Consider the simple case of a base 2 floating-point system with 3-bit significands (hence two bits of fraction). The exact result of 1.11×2^{-2} multiplied by $1.11 \times 2^{E_{\min}}$ is $0.110001 \times 2^{E_{\min}}$. If the rounding mode is round toward plus infinity, the rounded result is the normal number $1.00 \times 2^{E_{\min}}$. Should underflow be signaled? Signaling underflow means that one is using the *before rounding* rule, because the result was denormal before rounding. Not signaling underflow means that one is using the *after rounding* rule, because the result is normalized after rounding. The IEEE standard provides for choosing either rule; however, the one chosen must be used consistently for all operations.

As mentioned in the addition section, the trap handler, if there is one, should be called whenever the result is denormal. If there is no trap handler, the underflow exception is signaled only when the result is denormal and inexact. Normally, inexact means there was a result that couldn't be represented exactly and had to be rounded. Consider again the example of $(1.11 \times 2^{-2}) \times (1.11 \times 2^{E_{\min}}) = 0.110001 \times 2^{E_{\min}}$, with round to nearest in effect. The delivered result is $0.11 \times 2^{E_{\min}}$, which had to be rounded, causing inexact to be signaled. But is it correct to also signal underflow? Gradual underflow loses significance because the exponent range is bounded. If the exponent range were unbounded, the delivered result would be $1.10 \times 2^{E_{\min}-1}$, exactly the same answer obtained with gradual underflow. The fact that denormalized numbers have fewer bits in their significand than normalized numbers therefore doesn't make any difference in this case. The commentary to the standard [Cody et al. 1984] encourages this as the criterion for setting the underflow flag. That is, it should be set whenever the delivered result is different from what would be delivered in a system with the same fraction size, but with a very large exponent range. However, owing to the difficulty of implementing this scheme, the standard allows setting the underflow flag whenever the result is denormal and different from the infinitely precise result.

Precision of Multiplication

In the discussion of integer multiplication, we mentioned that designers must decide whether to deliver the low-order word of the product or the entire product. A similar issue arises in floating-point multiplication, where the exact product can be rounded to the precision of the operands or to the next higher precision. In the case of integer multiplication, none of the standard high-level languages contains a construct that would generate a "single times single gets double" instruction. The situation is different for floating point. Not only do many languages allow assigning the product of two single-precision variables to a double-precision one, but the construction can also be exploited by numerical algorithms. The best-known case is using iterative refinement to solve linear systems of equations.

A.6 Division and Remainder

Iterative Division

We earlier discussed an algorithm for integer division. Converting it into a floating-point division algorithm is similar to converting the integer multiplication algorithm into floating point. If the numbers to be divided are $s_1 2^{e_1}$ and $s_2 2^{e_2}$ then the divider will compute s_1/s_2, and the final answer will be this quotient multiplied by $2^{e_1-e_2}$. Referring to Figure A.2(b) (page A-4), the alignment of operands is slightly different from integer division. Load s_2 into b and $s_1/2$ into P so that s_1 is shifted right one bit. Then the integer algorithm for division can be used, and the result will be of the form $q_0.q_1\cdots$. For floating-point division, the A register is not needed to hold the operands. To round, simply compute two additional quotient bits (guard and round) and use the remainder as the sticky bit. The guard digit is necessary because the first quotient bit might be zero. However, since the numerator and denominator are both normalized, it is not possible for the two most significant quotient bits to be zero.

There is a different approach to division, based on iteration. An actual machine that uses this algorithm will be discussed in Section A.10. First, we will describe the two main iterative algorithms and then discuss the pros and cons of iteration compared to the direct algorithms. There is a general technique for constructing iterative algorithms, called *Newton's iteration*, shown in Figure A.9.

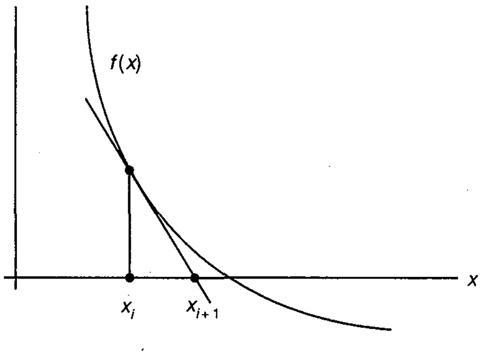


FIGURE A.9 Newton's iteration for zero finding. If x_i is an estimate for a zero of f, then x_{i+1} is a better estimate. To compute x_{i+1}, find the intersection of the x axis with the tangent line to f at x_i.

First, cast the problem in the form of finding the zero of a function. Then, starting from a guess for the zero, approximate the function by its tangent at that guess and form a new guess based on where the tangent has a zero. If x_i is a guess at a zero, then the tangent line has the equation

$$y - f(x_i) = f'(x_i)(x - x_i)$$

This equation has a zero at

A.6.1 $$x = x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

To recast division as finding the zero of a function, consider $f(x) = 1/x - b$. Since the zero of this function is at $1/b$, applying Newton's iteration to it will give an iterative method of computing $1/b$ from b. Using $f'(x) = -1/x^2$, Equation A.6.1 becomes

A.6.2 $$x_{i+1} = x_i - \frac{1/x_i - b}{-1/x_i^2} = x_i + x_i - x_i^2 b = x_i(2 - x_i b)$$

Thus, we could implement computation of a/b using the following method:

1. Scale b to lie in the range $1 \le b < 2$ and get an approximate value of $1/b$ (call it x_0) using a table lookup.
2. Iterate $x_{i+1} = x_i(2 - x_i b)$ until reaching an x_n that is accurate enough.
3. Compute ax_n and reverse the scaling done in step 1.

Here are some more details. How many times will step 2 have to be iterated? To say that x_i is accurate to p bits means that $(x_i - 1/b)/(1/b) = 2^{-p}$, and a simple algebraic manipulation shows $(x_{i+1} - 1/b)/(1/b) = 2^{-2p}$. Thus the number of correct bits doubles at each step. Newton's iteration is **self-correcting** in the sense that making an error in x_i doesn't really matter. That is, it treats x_i as a guess at $1/b$ and returns x_{i+1} as an improvement on it (roughly doubling the digits). One thing that would cause x_i to be in error is rounding error. More importantly, however, in the early iterations we can take advantage of the fact that we don't expect many correct bits by performing the multiplication in reduced precision, thus gaining speed without sacrificing accuracy. Some other applications of Newton's iteration are discussed in the Exercises.

The second iterative division method is sometimes called *Goldschmidt's algorithm*. It is based on the idea that to compute a/b, you should multiply the numerator and denominator by a number r with $rb \approx 1$. In more detail, let $x_0 = a$ and $y_0 = b$. At each step compute $x_{i+1} = r_i x_i$ and $y_{i+1} = r_i y_i$. Then the quotient $x_{i+1}/y_{i+1} = x_i/y_i = a/b$ is constant. If we pick r_i so that $y_i \to 1$, then $x_i \to a/b$, so the x_i converge to the answer we want. This same idea can be used to compute other functions. For example, to compute the square root of a, let $x_0 = a$ and $y_0 = a$, and at each step compute $x_{i+1} = r_i^2 x_i$, $y_{i+1} = r_i y_i$. Then $x_{i+1}/y_{i+1}^2 = x_i/y_i^2 = 1/a$, so if the r_i are chosen to drive $x_i \rightarrow 1$, then $y_i \rightarrow \sqrt{a}$. This technique is used to compute square roots on the TI 8847.

Returning to Goldschmidt's division algorithm, set $x_0 = a$ and $y_0 = b$, and write $b = 1 - \delta$, where $|\delta| < 1$. If we pick $r_0 = 1 + \delta$, then $y_1 = r_0 y_0 = 1 - \delta^2$. We next pick $r_1 = 1 + \delta^2$, so that $y_2 = r_1 y_1 = 1 - \delta^4$, and so on. Since $|\delta| < 1$, $y_i \rightarrow 1$. With this choice of r_i, the x_i will be computed as $x_{i+1} = r_i x_i = (1 + \delta^{2^i})x_i = (1 + (1 - b)^{2^i})x_i$, or

A.6.3 $$x_{i+1} = a[1 + (1 - b)][1 + (1 - b)^2][1 + (1 - b)^4]\cdots[1 + (1 - b)^{2^i}]$$

There appear to be two problems with this algorithm. First, convergence is slow when b is not near 1 (that is, δ is not near 0); and second, the formula isn't self-correcting—since the quotient is being computed as a product of independent terms, an error in one of them won't get corrected. To deal with slow convergence, if you want to compute a/b, look up an approximate inverse to b (call it b'), and run the algorithm on ab'/bb'. This will converge rapidly since $bb' \approx 1$.

To deal with the self-correction problem, the computation should be run with a few bits of extra precision to compensate for rounding errors. However, Goldschmidt's algorithm does have a weak form of self-correction, in that the precise value of the r_i does not matter. Thus, in the first few iterations, you can choose r_i to be a truncation of $1 + \delta^{2^i}$ which may make these iterations run faster without affecting the speed of convergence. If r_i is truncated, then y_i is no longer exactly $1 - \delta^{2^i}$, so Equation A.6.3 can no longer be used, but it is easy to organize the computation so that it does not depend on the precise value of r_i. With these changes, Goldschmidt's algorithm is as follows (the notes in brackets show the connection with our earlier formulas).

1. Scale a and b so that $1 \le b < 2$.
2. Look up an approximation to $1/b$ (call it b') in a table.
3. Set $x_0 = ab'$ and $y_0 = bb'$.
4. Iterate until x_i is close enough to a/b :

```
$r \approx 2 - y$           [if $y_i = 1 + \delta_i$, then $r \approx 1 - \delta_i$]
$y = y \times r$            [$y_{i+1} = y_i \times r \approx 1 - \delta_i^2$]
$x = x \times r$            [$x_{i+1} = x_i \times r$]
```

The two iteration methods are related. Suppose in Newton's method that we unroll the iteration and compute each term x_{i+1} directly in terms of b, instead of recursively in terms of x_i. By carrying out this calculation, we discover that

$$x_{i+1} = x_0(2 - x_0 b)(1 + (x_0 b - 1)^2)(1 + (x_0 b - 1)^4)\cdots(1 + (x_0 b - 1)^{2^i})$$

This formula is of a very similar form to Equation A.6.3 when $a = 1$. In fact, if the iterations were done to infinite precision, the two methods would yield exactly the same sequence x_i.

The advantage of iteration is that it doesn't require special divide hardware, but can instead use the multiplier (which, however, requires extra control). Further, on each step, it delivers twice as many digits as in the previous step—unlike ordinary division, which produces a fixed number of digits at every step. There are two disadvantages with inverting by iteration. The first is that the IEEE standard requires division to be correctly rounded, but iteration only delivers a result that is close to the correctly rounded answer. In the case of Newton's iteration, which computes $1/b$ instead of a/b directly, there is an additional problem. Even if $1/b$ was correctly rounded, there is no guarantee that a/b will be. Take 5/7 as an example: To two digits of accuracy 1/7 is 0.14, and 5×0.14 is 0.70, but 5/7 is 0.71. The second disadvantage is that iteration does not give a remainder. This is especially troublesome if the floating-point divide hardware is being used to perform integer division, since a remainder operation is present in almost every high-level language.

Traditional folklore has held that the way to get a correctly rounded result from iteration is to compute $1/b$ to slightly more than $2p$ bits, compute a/b to slightly more than $2p$ bits, and then round to p bits. However, there is a faster way, which apparently was first implemented on the TI 8847. In this method, a/b is computed to about six extra bits of precision, giving a preliminary quotient q. By comparing qb with a (again with only six extra bits), it is possible to quickly decide whether q is correctly rounded or whether it needs to be bumped up or down by 1 in the least significant place. This algorithm is explored further in the Exercises.

One factor to take into account when deciding on division algorithms is the relative speed of division and multiplication. Since division is more complex than multiplication, it will run more slowly. As a general rule of thumb, division algorithms should try to achieve a speed that is about one-third that of multiplication. One argument in favor of this rule is that there are real programs (such as some versions of Spice) where the ratio of division to multiplication is 1:3. Another place where a factor of three arises is in the standard iterative method for computing square root. This method involves one division per iteration, but can be replaced by one using three multiplications. This is discussed in the Exercises.

Floating-Point Remainder

For nonnegative integers, integer division and remainder satisfy

$$a = (a \text{ DIV } b)b + a \text{ REM } b, \quad 0 \le a \text{ REM } b < b$$

A floating-point remainder x REM y can be similarly defined as $x = \text{INT}(x/y)y +$ x REM y. How should x/y be converted to an integer? The IEEE remainder function uses the round-to-even rule. That is, pick $n = \text{INT}\ (x/y)$ so that $|x/y - n| \leq 1/2$. If two different n satisfy this relation, pick the even one. Then REM is defined to be $x - yn$. Unlike integers where $0 \leq a$ REM $b < b$, for floating-point numbers $|x \text{ REM } y| \leq y/2$. Although this defines REM precisely, it is not a practical operational definition, because n can be huge. In single precision, n could be as large as $2^{127}/2^{-126} = 2^{253} \approx 10^{76}$.

There is a natural way to compute REM if a direct division algorithm is used. Proceed as if you were computing x/y. If $x = s_1 2^{e_1}$ and $y = s_2 2^{e_2}$ and the divider is as in Figure A.2(b) (page A-4), then load s_1 into P and s_2 into B. After $e_1 - e_2$ division steps, the P register will hold a number r of the form $x - yn$ satisfying $0 \leq r < y$. The IEEE remainder is then either r or $r - y$. It is only necessary to keep track of the last quotient bit produced, which is needed in order to resolve halfway cases. Unfortunately, $e_1 - e_2$ can be a lot of steps, and floating-point units typically have a maximum amount of time they are allowed to spend on one instruction. Thus, it is usually not possible to implement REM directly. None of the chips discussed in Section A.10 implement REM, but they could by providing a remainder-step instruction—this is what is done on the Intel 8087 family. A remainder step takes as arguments two numbers x and y, and performs divide steps until either the remainder is in P, or else n steps have been performed, where n is a small number, such as the number of steps required for division in the highest supported precision. The REM driver calls the REM-step instruction $\lfloor (e_1 - e_2)/n \rfloor$ times, initially using x as the numerator, but then replacing it with the remainder from the previous REM step. It is useful if the REM-step instruction returns the low-order three bits of the quotient, since when doing trigonometric argument reduction to the interval $(0, \pi/4)$, you need to know the value of n mod 8 in order to know what quadrant you are in.

Currently, most of the fastest floating-point chips don't implement remainder, even though it is a required part of the IEEE standard. Since the standard allows implementations to be a combination of hardware and software, the REM operation could be implemented entirely in software. However, availability of the REM-step instruction would make computing REM much simpler. Is a REM-step instruction worth it? For two reasons this situation is difficult to decide on the basis of frequency data. First, because REM is peculiar to the IEEE standard, few people are currently using it. Testing the demand for REM is somewhat like trying to estimate the demand for a new product. Second, the main benefit from REM is not an increase in performance, but rather an increase in accuracy, and it is not easy to quantify the value of accuracy. What we will do here is simply present the primary application of REM, which is argument reduction for periodic functions, like sin and cos.

There are some subtle issues involved in argument reduction. To simplify things, imagine that we are working in base 10 with 5 significant figures, and consider computing $\sin x$. Suppose that $x = 7$. Then we reduce by $\pi = 3.1416$

and compute sin(7) = sin(7 − 2×3.1416) = sin(0.7168) instead. But suppose we want to compute sin(2.0 × 10^5). Then 2 × 10^5/3.1416 = 63661.8, which in our 5-place system comes out to be 63662. Since multiplying 3.1416 times 63662 gives 200000.5392, which rounds to 2.0000 × 10^5, argument reduction reduces 2 × 10^5 to 0, which is not even close to being correct. The problem is that our 5-place system does not have the precision to do correct argument reduction. Suppose we had the REM operator. Then we could compute 2 × 10^5 REM 3.1416 and get −.5392. However, this is still not correct because we used 3.1416, which is an approximation for π. The value of 2 × 10^5 REM π is −.071513. The difficulty is that we subtracted two nearby numbers, 2 × 10^5 and 63662×3.1416, where 63662×3.1416 was slightly in error due to approximating π. Even though REM has the effect of performing the subtraction exactly, all the significant figures in 63662 × 3.1416 canceled, leaving behind only rounding error.

Traditionally, there have been two approaches to computing periodic functions with large arguments. The first is to return an error for their value when x is large. The second is to store π to a very large number of places and do exact argument reduction. The REM operator is not much help in either of these situations. There is a third approach that has been used in some math libraries, such as the Berkeley UNIX 4.3bsd release. In these libraries, π is computed to the nearest floating-point number. Let's call this machine π, and denote it by π'. Then when computing $\sin x$, reduce x using x REM π'. As we saw in the above example, x REM π' is quite different from x REM π, so that computing $\sin x$ as sin(x REM π') will not give the exact value of $\sin x$. However, computing trigonometric functions in this fashion has the property that all familiar identities (such as $\sin^2 x + \cos^2 x = 1$) are true to within a few rounding errors. Thus, using REM together with machine π provides a simple method of computing trigonometric functions that is accurate for small arguments and still useful for large arguments in most applications.

A.7 Precisions and Exception Handling

Precisions

Implementations of the IEEE standard are only required to support single precision. Thus, the computer designer must make a choice about what other precisions to support. Because of the widespread use of double precision in scientific computing, double precision is almost always implemented.

Double-extended precision is more problematic. Although the Motorola 68882 and Intel 387 coprocessors implement extended precision, most of the more recently designed, high-performance floating-point chips do not implement extended precision. Among the reasons are that the 80-bit width of extended precision is awkward for 64-bit buses and registers, and that many high-level languages do not give the user access to extended precision. However, extended precision is very useful to writers of mathematical software. As an example, consider writing a library routine to compute the length of a vector in the plane $\sqrt{x^2 + y^2}$. If x is larger than $2^{E_{\max}/2}$, then computing this in the obvious way will overflow. This means that either the allowable exponent range for this subroutine will be cut in half, or a more complex algorithm using scaling will have to be employed. But if extended precision is available, then the simple algorithm will work. Computing the length of a vector is a simple task, and it is not difficult to come up with an algorithm that doesn't overflow. However, there are more complex problems for which extended precision means the difference between a simple, fast algorithm and a much more complex one. One of the best examples of this is binary/decimal conversion. An efficient algorithm for binary-to-decimal conversion that makes essential use of extended precision is very readably presented in Coonen [1984]. This algorithm is also briefly sketched in Goldberg [1989]. Computing accurate values for transcendental functions is another example of a problem that is made much easier if extended precision is present.

One very important fact about precision concerns *double rounding*. To illustrate in decimal, suppose that we want to compute 1.9×0.66, and that single precision is two digits, while extended precision is three digits. The exact result of the product is 1.254. Rounded to extended precision, the result is 1.25. When further rounded to single precision, we get 1.2. However, the result of 1.9×0.66 correctly rounded to single precision is 1.3. Thus, rounding twice may not produce the same result as rounding once. Suppose you want to build hardware that only does double-precision arithmetic. Can you simulate single precision by computing first in double precision and then rounding to single? The above example suggests that you can't. However, double rounding is not always dangerous. In fact, the following rule is true (although it is not easy to prove).

> *If x and y have p-bit significands, and $x + y$ is computed exactly and then rounded to q places, a second rounding to p places will not change the answer if $p \le (q-1)/2$. This is true not only for addition, but also for multiplication, division, and square root.*

In our example above, $q = 3$, and $p = 2$, so $2 \le (3 - 1)/2$ is not true. On the other hand, for IEEE arithmetic, double precison has $p = 53$, and single precision is $p = 24 \le (q-1)/2 = 26$. Thus, single precision can be implemented by computing in double precision (that is, computing the answer exactly and then rounding to double) and then rounding to single precision.

The standard requires implementations to provide versions of addition, subtraction, multiplication, division, and remainder that take two operands of the same precision and produce a result of that precision. It also recommends that implementations allow operations that take operands of two different precisions and return a result whose precision is at least as wide as the widest operand. The standard allows implementations to combine two operands and return a result in a higher precision. Remember that the result of an operation is the exact result rounded to the destination precision. What the standard does not allow is combining two operands and returning a result in a lower precision. Although at first this may seem like a minor restriction, consider again the problem of computing $\sqrt{x^2 + y^2}$. If x and y are double, then you might like to compute $x^2 + y^2$ in extended precision and then compute a square root that takes an extended-precision argument and returns a double-precision answer. But this is not allowed by the standard.

There is a related issue. The standard permits combining two extended variables to produce a result that is stored in extended format, but rounded to double precision. However, this doesn't help in the square root example, because the result of the square root must still be explicitly converted from an extended format to a double-precision format.

Exceptions

The IEEE standard defines five exceptions: underflow, overflow, divide by zero, inexact, and invalid. By default, when these exceptions occur, they merely set a flag and the computation continues. The flags are *sticky*, meaning that once set they remain set until explicitly cleared. The standard strongly encourages implementations to provide a trap-enable bit for each exception. When an exception with an enabled trap handler occurs, a user trap handler is called, and the value of the associated exception flag is undefined.

The underflow, overflow, and divide-by-zero exceptions are found in most other systems. The *inexact exception* is peculiar to IEEE arithmetic and occurs when either the result of an operation must be rounded or when it overflows. In fact, since 1/0 and an operation that overflows both deliver ∞, the exception flags must be consulted to distinguish between them. The inexact exception is an unusual "exception," in that it is not really an exceptional condition because it occurs so frequently. Thus, enabling a trap handler for inexact will most likely have a severe impact on performance. The *invalid exception* is for things like $\sqrt{-1}$, 0/0 or $\infty - \infty$, which don't have any natural value as a floating-point number or as $\pm\infty$. Thus, 1/0 causes a divide by zero exception and delivers ∞, whereas 0/0 causes an invalid exception and delivers a NaN. There is a twist in IEEE underflow, because it is not always signaled when numbers fall below $1.0 \times 2^{E_{\min}}$. If a user trap handler is not installed, then underflow is signaled only if the result of an operation is below $2^{E_{\min}}$ and is inexact.

The IEEE standard assumes that when a trap occurs, it is possible to identify the operation that trapped and its operands. On machines with pipelining, or machines with multiple arithmetic units, when an exception occurs, it may not be enough to simply have the trap handler examine the program counter. Hardware support may be necessary in order to identify exactly which operation trapped. Another problem is illustrated by the following program fragment.

```
X = Y * Z;
Z = A + B;
```

These two instructions might well be executed in parallel. If the multiply traps, its argument Z could already have been overwritten by the addition, especially since addition is usually faster than multiplication. Computer systems that support trapping in the IEEE standard must provide some way to save the value of Z, either in hardware or by having the compiler avoid such a situation in the first place.

One approach to this problem, used in the MIPS R3010, is to treat floating-point exceptions similarly to page-fault exceptions. If an instruction that assigns a memory location to a register causes a page fault, the execution of the instruction must stall before it clobbers the register because (for example) that very register might be used to reference the memory that faulted. The key to making this work is that the memory address is computed early in the instruction cycle, before the instruction actually writes anything. A similar trick can be done with floating-point operations. An instruction that may cause an exception can be identified early in the instruction cycle. For example, an addition can overflow only if one of the operands has an exponent of $E_{\max}$, and so on. This early check is conservative: It might flag an operation that doesn't actually cause an exception. However, if such false positives are rare, then this technique will have excellent performance. When an instruction is tagged as being possibly exceptional, special code in a trap handler can compute it without destroying any state. Remember that all these problems occur only when trap handlers are enabled. Otherwise, setting the exception flags during normal processing is straightforward.

There is a subtlety that should be mentioned that involves the underflow trap. When there is no underflow trap handler, the result of an operation that involves an underflow is a denormal number. When there is a trap handler, it is provided with the result of the operation with the exponent wrapped around. Now there is a potential double-rounding problem. If the rounding mode is round toward nearest, when there is a trap handler the result is correctly rounded to p significant bits. If there is no trap handler, the result is rounded to less than p bits, depending on how many leading zeros the denormal number has. If the trap handler wants to return the denormal result, it can't just round its argument, because that might lead to a double-rounding error. Thus, the trap handler must be passed at least one extra bit of information if it is to be able to deliver the correctly rounded result.

A.8 Speeding Up Integer Addition

The previous section showed that there are many steps that go into implementing floating-point operations. However, each floating-point operation eventually reduces to an integer operation. Thus, increasing the speed of integer operations will also lead to faster floating point.

Integer addition is the simplest operation and the most important. Even for programs that don't do explicit arithmetic, addition must be performed to increment the program counter and to do address calculations. Despite the simplicity of addition, there isn't a single best way to perform high-speed addition. We will discuss three techniques that are in current use: carry lookahead, carry skip, and carry select.

Carry Lookahead

An *n*-bit adder is just a combinational circuit. It can therefore be written by a logic formula whose form is a sum of products and can be computed by a circuit with two levels of logic. How does one figure out what this circuit looks like? Recall from Equation A.2.1 that the formula for the *i*th sum bit is

$$s_i = a_i\overline{b_i}\,\overline{c_i} + \overline{a_i}b_i\overline{c_i} + \overline{a_i}\,\overline{b_i}c_i + a_ib_ic_i \tag{A.8.1}$$

The problem with this formula is that although we know the values of a_i and b_i—they are inputs to the circuit—we don't know c_i. So our goal is to write c_i in terms of a_i and b_i. To accomplish this, we first rewrite Equation A.2.2 (page A-2) as

$$c_{i+1} = g_i + p_ic_i, \quad g_i = a_ib_i, \quad p_i = a_i + b_i \tag{A.8.2}$$

Here is the reason for the symbols *p* and *g*: If g_i is true, then c_{i+1} is certainly true, so a carry is *generated.* Thus, *g* is for generate. If p_i is true, then if c_i is true, it is *propagated* to c_{i+1}. Start with Equation A.8.1 and use Equation A.8.2 to replace c_i with $g_{i-1} + p_{i-1}c_{i-1}$. Then, use Equation A.8.2 with $i - 1$ in place of *i*, to replace c_{i-1} with c_{i-2}, and so on. This gives the result

$$c_{i+1} = g_i + p_i\,g_{i-1} + p_ip_{i-1}g_{i-2} + \cdots + p_i\,p_{i-1}\cdots p_1\,g_0 + p_ip_{i-1}\cdots p_1p_0c_0 \tag{A.8.3}$$

An adder that computes carries using Equation A.8.3 is called a *carry-lookahead adder,* or CLA adder. A CLA adder requires one logic level to form *p* and *g,* two levels to form the carries, and two for the sum, for a grand total of five logic levels. This is a vast improvement over the 2*n* levels required for the ripple-carry adder.

Unfortunately, as is evident from Equation A.8.3 or from Figure A.10, a carry-lookahead adder on *n* bits requires a fan-in of $n+1$ at the OR gate as well as at the rightmost AND gate. Also, the p_{n-1} signal must drive *n* AND gates. In addition, the rather irregular structure and many long wires of Figure A.10 make it impractical to build a full carry-lookahead adder when *n* is large.

However, we can use the carry-lookahead idea to build an adder that has about $\log_2 n$ logic levels (substantially less than the 2*n* required by a ripple-carry adder), and yet has a simple, regular structure. The idea is to build up the *p*'s and *g*'s in steps. We have already seen that

$$c_1 = g_0 + c_0p_0$$

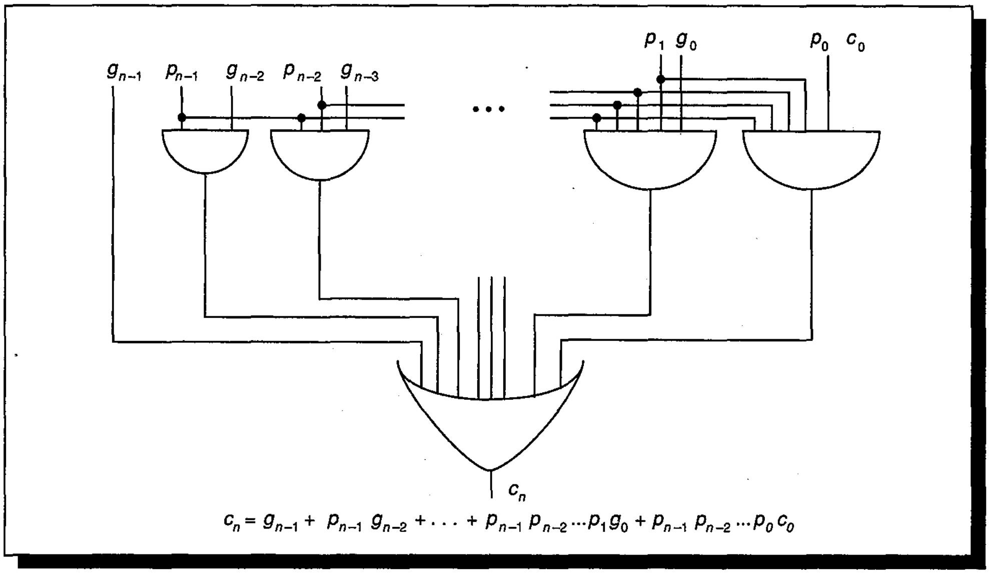

g_{n-1}
p_{n-1}
g_{n-2}
p_{n-2}
g_{n-3}
p_1 g_0
p_0 c_0
c_n
$c_n = g_{n-1} + p_{n-1}\, g_{n-2} + \cdots + p_{n-1}\, p_{n-2} \cdots p_1 g_0 + p_{n-1}\, p_{n-2} \cdots p_0 c_0$

FIGURE A.10 Pure carry-lookahead circuit for computing the carry out c_n of an n-bit adder.

This says there is a carry out of the 0th position (c_1) if there is either a carry generated in the 0th position, or if there is a carry into the 0th position and the carry propagates. Similarly,

$$c_2 = G_{01} + P_{01}c_0$$

G_{01} means there is a carry generated out of the block consisting of the first two bits. P_{01} means that a carry propagates through this block. P and G have the following logic equations:

$$\begin{aligned} G_{01} &= g_1 + p_1 g_0 \\ P_{01} &= p_1 p_0 \end{aligned}$$

More generally, for any j with $i < j$, $j + 1 < k$, we have the recursive relations

$$c_{k+1} = G_{ik} + P_{ik}c_i \tag{A.8.4}$$

$$G_{ik} = G_{j+1,k} + P_{j+1,k}G_{ij} \tag{A.8.5}$$

$$P_{ik} = P_{ij}P_{j+1,k} \tag{A.8.6}$$

Equation A.8.5 says that a carry is generated out of the block consisting of bits i through k inclusive if it is generated in the high-order part of the block (j + 1, k) or if it is generated in the low-order (i,j) part of the block and then propagated through the high part. These equations will also hold for $i \leq j < k$ if we set $G_{ii} = g_i$ and $P_{ii} = p_i$.

Example: Express P_{03} and G_{03} in terms of p's and g's.

Answer: Using A.8.6, $P_{03} = P_{01}P_{23} = P_{00}P_{11}P_{22}P_{33}$. Since $P_{ii} = p_i$, $P_{03} = p_0p_1p_2p_3$. For G_{03}, Equation A.8.5 says $G_{03} = G_{23} + P_{23}G_{01} = (G_{33} + P_{33}G_{22}) + (P_{22}P_{33})(G_{11} + P_{11}G_{00}) = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0$.

With these preliminaries out of the way, we can now show the design of a practical CLA adder. The adder consists of two parts. The first part computes various values of P and G from p_i and g_i, using Equations A.8.5 and A.8.6; the second part uses these P and G values to compute all the carries via Equation A.8.4. The first part of the design is in Figure A.11. At the top of the diagram, input numbers $a_7 \cdots a_0$ and $b_7 \cdots b_0$ are converted to p's and g's using cells of type 1. Then various P's and G's are generated by combining cells of type 2 in a binary-tree structure. The second part of the design is shown in Figure A.12. By feeding c_0 in at the bottom of this tree, all the carry bits come out the top. Each cell must know a pair of (P,G) values in order to do the conversion, and the value it needs is written inside the cells. Now compare Figure A.11 and Figure A.12. There is a one-to-one correspondence between cells, and the value of (P,G) needed by the carry-generating cells is exactly the value known by the

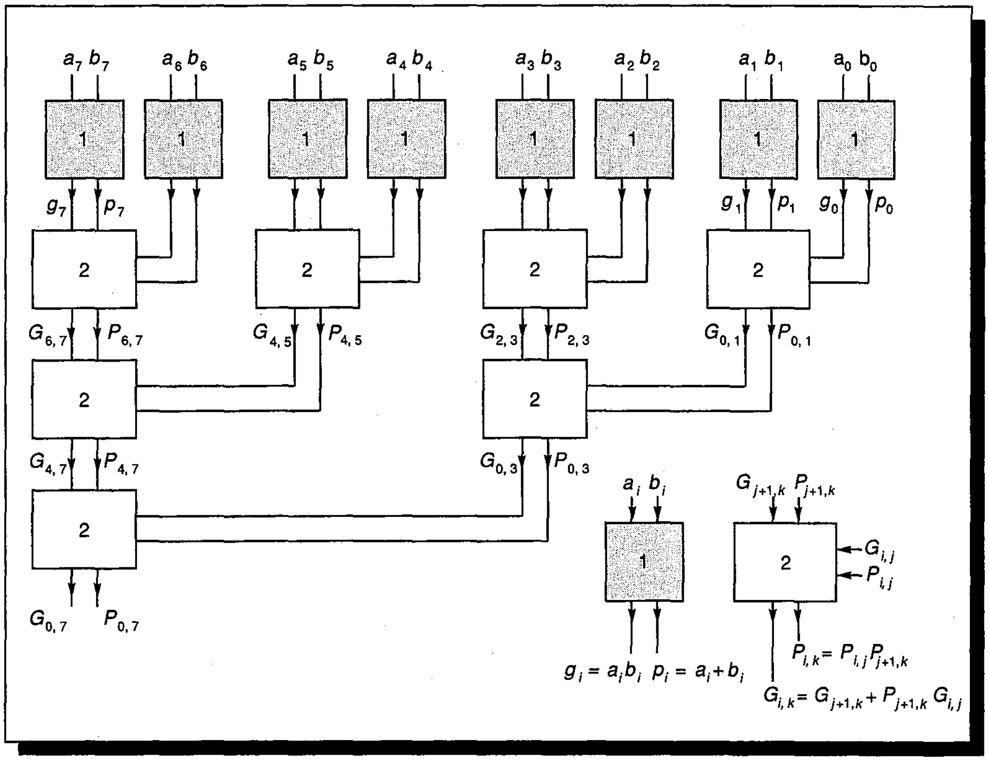


FIGURE A.11 First part of carry-lookahead tree. As signals flow from the top to the bottom, various values of P and G are computed.

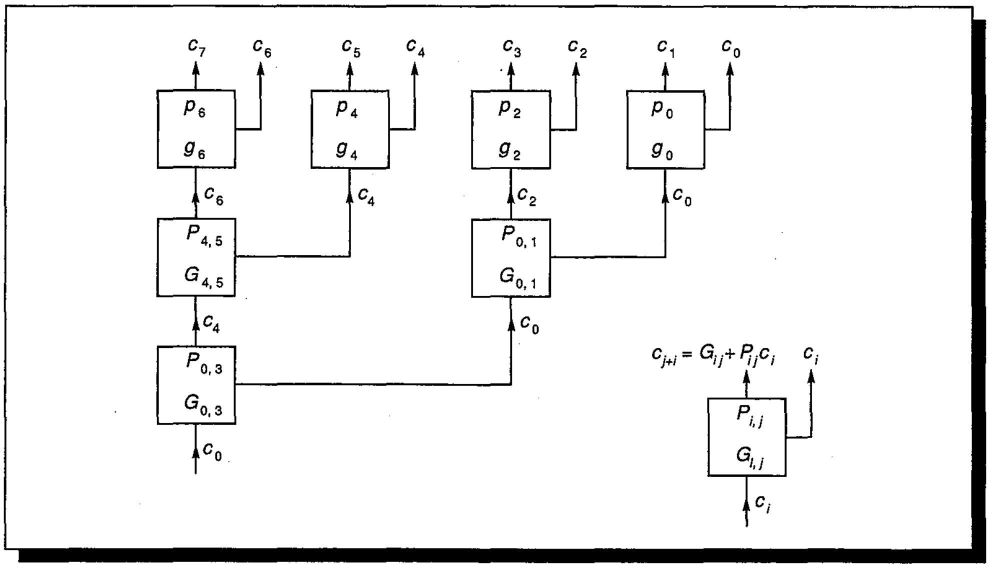


FIGURE A.12 Second part of carry-lookahead tree. Signals flow from the bottom to the top, combining with *P* and *G* to form the carries.

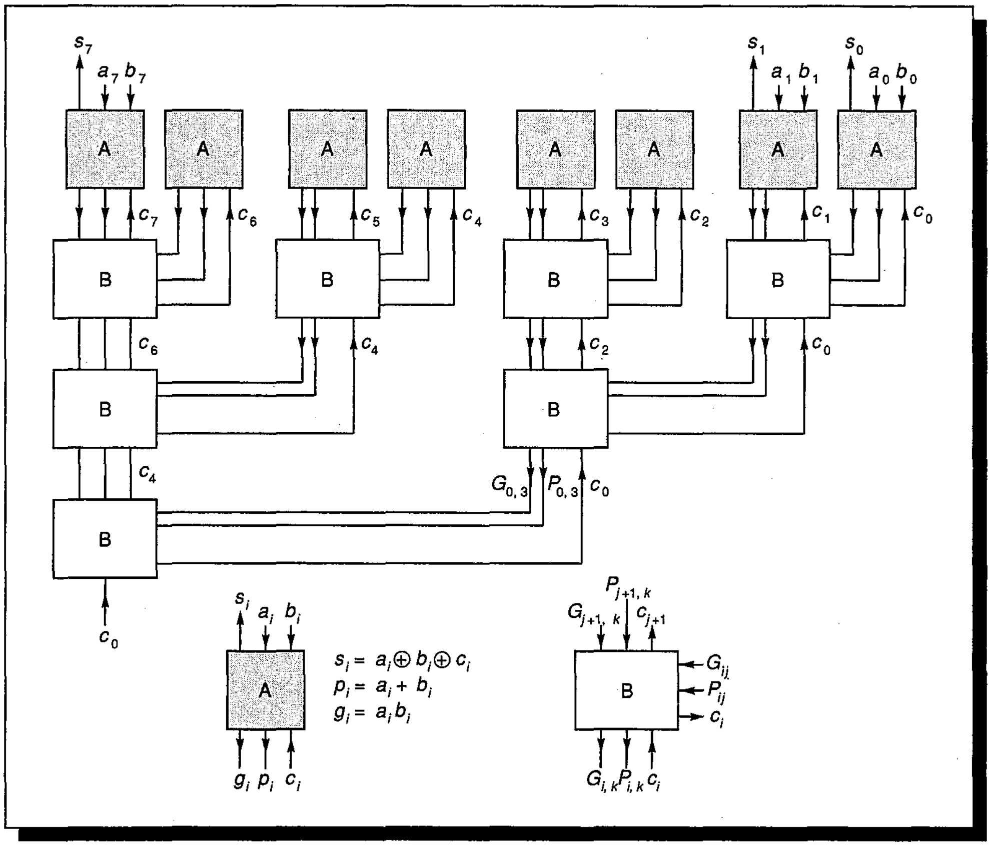


FIGURE A.13 Complete carry-lookahead tree adder. This is the combination of Figures A.11 and A.12. The numbers to be added enter at the top, flow to the bottom to combine with c_0, and then flow back up to compute the sum bits.

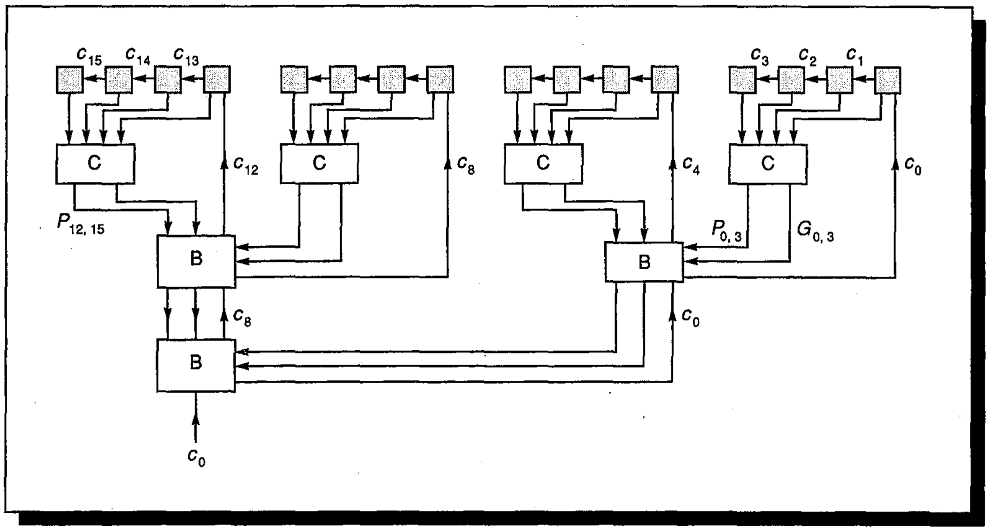


FIGURE A.14 Combination of CLA adder and ripple-carry adder. In the top row, carries ripple within each group of four boxes.

corresponding (*P*,*G*) generating cells. The combined cell is shown in Figure A.13. The numbers to be added flow into the top and downward through the tree, combining with c_0 at the bottom and flowing back up the tree to form the carries. Note that there is one thing missing from Figure A.13: a small piece of extra logic to compute c_8 for the carry out of the adder.

The bits in a CLA must pass through about $\log_2 n$ logic levels, compared with $2n$ for a ripple-carry adder. This is a substantial speed improvement, especially for a large *n*. Whereas the ripple-carry adder had *n* cells, however, the CLA adder has $2n$ cells, although in our layout they will take $n \log n$ space. The point is that a small investment in size pays off in a dramatic improvement in speed.

There are a number of technology-dependent modifications that can improve CLA adders. For example, if each node of the tree has three inputs instead of two, then the height of the tree will decrease from $\log_2 n$ to $\log_3 n$. Of course, the cells will be more complex and thus might operate more slowly, negating the advantage of the decreased height. For technologies where rippling works well, a hybrid design might be better. This is illustrated in Figure A.14. Carries ripple between adders at the top level, while the "B" boxes are the same as in Figure A.13. This design will be faster if the time to ripple between four adders is faster than the time it takes to traverse a level of "B" boxes.

Carry-Skip Adders

A *carry-skip adder* sits midway between a ripple-carry adder and a carry-lookahead adder, both in terms of speed and cost. (A carry-skip adder is not called a CSA, as that name is reserved for carry-save adders.) The motivation for this adder comes from examining the equations for *P* and *G*. For example,

$$P_{03} = p_0 p_1 p_2 p_3$$

$$G_{03} = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0$$

Computing P is much simpler than computing G, and a carry-skip adder only computes the P's. Such an adder is illustrated in Figure A.15. Carries begin rippling simultaneously through each block. If any block generates a carry, then the carry out of a block will be true, even though the carry in to the block may not be correct yet. If at the start of each add operation the carry in to each block is zero, then no spurious carry outs will be generated. Thus, the carry out of each block can thus be thought of as if it were the G signal. Once the carry out from the least significant block is generated, it not only feeds into the next block, but is also fed through the AND gate with the P signal from that next block. If the carry out and P signals are both true, then the carry **skips** the second block and is ready to feed into the third block, and so on. The carry-skip adder is only practical if the carry in signals can be easily cleared at the start of each operation—for example by precharging in CMOS.

To analyze the speed of a carry-skip adder, let's assume that it takes one time unit for a signal to pass through two logic levels. Then it will take k time units for a carry to ripple across a block of size k, and it will take one time unit for a carry to skip a block. The longest signal path in the carry-skip adder starts with a carry being generated at the 0th position. Then it takes k time units to ripple through the first block, $n/k - 2$ time units to skip blocks, and k more to ripple through the last block. To be specific: If we have a 20-bit adder broken into groups of 4 bits, it will take 11 time units to perform an add. Suppose we keep the least significant block at 4 bits, but combine the next two blocks into a single 8-bit block. Then the time of the adder drops to 10 time units. However, if we had combined three blocks instead of two, then the time to ripple through this 3-block unit (12 bits in all) would dominate the time to add. However, the general principle is important: For a carry-skip adder, making the interior blocks larger will speed up the adder. In fact, the same idea of varying the block sizes can sometimes speed up other adder designs as well. Because of the large amount of rippling, a carry-skip adder is most appropriate for technologies where rippling is fast.

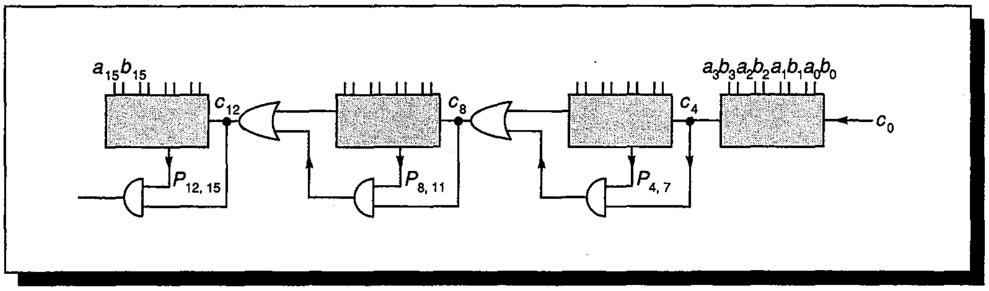


FIGURE A.15 Carry-skip adder.

Carry-Select Adder

A *carry-select adder* works on the following principle: Two additions are performed in parallel, one assuming the carry in is zero and the other assuming the carry in is one. When the carry in is finally known, the correct sum (which has been precomputed) is simply selected. An example of such a design is shown in Figure A.16. An 8-bit adder is divided into two halves, and the carry out from the lower half is used to select the upper half. If each block is computing its sum using rippling (a linear-time algorithm), then the design in Figure A.16 is twice

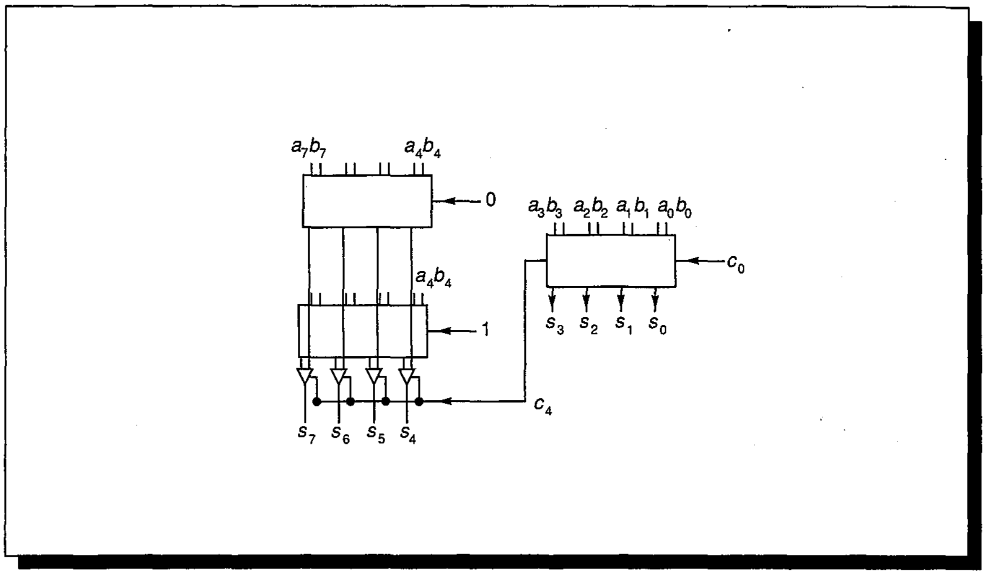
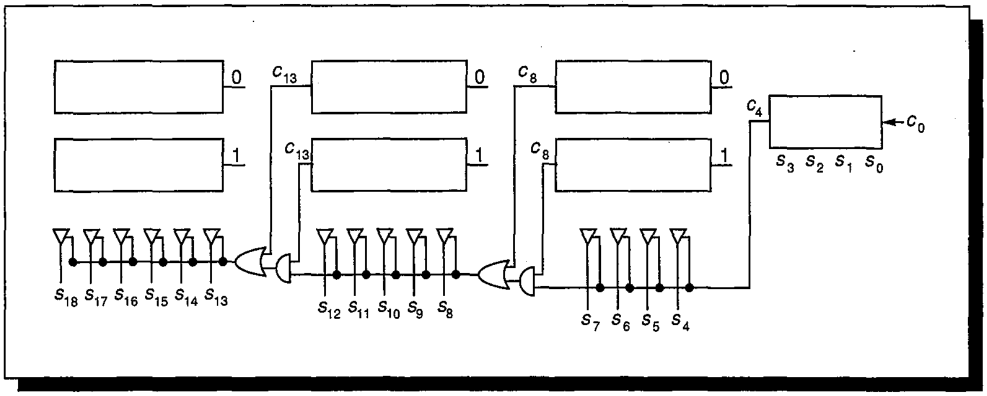


FIGURE A.16 Simple carry-select adder. At the same time that the sum of the low-order four bits are being computed, the high-order bits are being computed twice in parallel: once assuming that $c_4 = 0$, and once assuming $c_4 = 1$.



0
c_{13}
0
c_8
0
c_4
c_0
1
c_{13}
1
c_8
1
s_3 s_2 s_1 s_0
s_{18} s_{17} s_{16} s_{15} s_{14} s_{13}
s_{12} s_{11} s_{10} s_9 s_8
s_7 s_6 s_5 s_4

FIGURE A.17 Carry-select adder. As soon as the carry out of the rightmost block is known, it is used to select the other sum bits.

as fast at 50% more cost. However, note that the c_4 signal must drive many muxes, which may be very slow in some technologies. Instead of dividing the adder into halves, it could be divided into quarters for a still further speedup. This is illustrated in Figure A.17. If it takes k time units for a block to add k-bit numbers, and if it takes one time unit to compute the mux input from the two carry-out signals, then for optimal operation each block should be one bit wider than the next, as shown in Figure A.17. Therefore, as in the carry-skip adder, the best design involves variable-sized blocks.

As a summary of this section, the asymptotic time and space requirements for the different adders are given in Figure A.18. These different adders shouldn't be thought of as disjoint choices, but rather as building blocks to be used in constructing an adder. The utility of these different building blocks is highly dependent on the technology used. For example, the carry-select adder works well when a signal can drive many muxes, and the carry-skip adder is attractive in technologies where signals can be cleared at the start of each operation. Knowing the asymptotic behavior of adders is useful in understanding them, but relying too much on that behavior is a pitfall. The reason is that asymptotic behavior is only important as n grows very large. But n for an adder is the bits of precision, and double precision today is the same as it was twenty years ago—about 53 bits. Although it is true that as computers get faster, computations get longer—and thus have more rounding error, which in turn requires more precision—this effect grows very slowly with time.

	Time	**Space**
Ripple	$O(n)$	$O(n)$
CLA	$O(\log n)$	$O(n \log n)$
Carry skip	$O(\sqrt{n})$	$O(n)$
Carry select	$O(\sqrt{n})$	$O(n)$

FIGURE A.18 Asymptotic time and space requirements for four different types of adders.

A.9 Speeding Up Integer Multiplication and Division

The multiplication and division algorithms presented in Section A.2 are fairly slow, producing one bit per cycle (although that cycle might be a fraction of the CPU instruction cycle time). In this section we discuss various techniques for higher performance multiplication and division.

Shifting Over Zeros

Shifting over zeros is a technique that is not currently used much, but is instructive to consider. It is distinguished by the fact that its execution time is operand dependent. Its lack of use is primarily attributable to its failure to offer enough speedup over bit-at-a-time algorithms. In addition, pipelining, synchronization with the CPU, and good compiler optimization are difficult with algorithms that run in variable time. In multiplication, the idea behind shifting over zeros is to add logic that detects when the low-order bit of the A register is zero (see Figure A.2(a)) and, if so, skip the addition step and proceed directly to the shift step—hence the term *shifting over zeros*. This technique becomes more useful if the number of zeros in the A operand can be increased. The Exercises discuss how well Booth recoding does in increasing zeros.

What about shifting for division? In nonrestoring division, an ALU operation (either an addition or subtraction) is performed at every step, so that there appears to be no opportunity for skipping an operation. But think about division this way: To compute a/b, subtract multiples of b from a, and then report how many subtractions were done. At each stage of the subtraction process the remainder must fit into the P register of Figure A.2(b) (page A-4). In the case when the remainder is a small positive number, you normally subtract b; but suppose instead you only shifted the remainder and subtracted b the next time. As long as the remainder was sufficiently small (its high-order bit 0), after shifting it still would fit into the P register, and no information would be lost. However, this method does require changing the way we keep track of the number of times b has been subtracted from a. This idea usually goes under the name of *SRT division*, for Sweeney, Robertson, and Tocher, who independently proposed algorithms of this nature. The main extra complication of SRT division is that the quotient bits cannot be determined immediately from the sign of P at each step, as it can be in ordinary nonrestoring division.

More precisely, to divide a by b where a and b are n-bit numbers, load a and b into the A and B registers, respectively, of Figure A.2 (page A-4).

1. If B has k leading zeros when expressed using n bits, shift all the registers left k bits. After this shift, since b has $n + 1$ bits, its most significant bit will be 0, and its second-most-significant bit will be 1.

```
2. For $i = 0, n - 1$ do
       If the top three bits of P are equal, set $q_i = 0$ and shift (P,A) one bit left.
       If the top three bits of P are not all equal and P is negative, set $q_i = \bar{1}$, shift (P,A) one bit left, and add B.
       Otherwise set $q_i = 1$, shift (P,A) one bit left, and subtract B
   Endloop
```

3. If the final remainder is negative, correct the remainder by adding B, and correct the quotient by subtracting 1 from q_0. Finally, the remainder must be shifted k bits right, where k is the initial shift.

A numerical example is given in Figure A.19. Although we are discussing integer division, it helps in explaining the algorithm to move the binary point from the right of the least significant bit to the left of the most significant bit. Thus if $n = 4$ and the operation is 9/4, the A register holds 0.1001 and (remembering that the B register has $n + 1$ bits), the B register holds 0.0100.

Since this changes the binary point in both the numerator and denominator, the quotient is not affected. The remainder being a two's complement number, a P register of 1.1110_2 represents $-1/8$. With this convention, the P register holds numbers satisfying $-1 \le P < 1$. The first step of the algorithm shifts b so that $b \ge 1/2$. As before, let r be the value of the (P,A) pair. Our rule for which ALU operation to perform is this: If $-1/4 \le r < 1/4$ (true whenever the top three bits of P are equal), then compute $2r$ by shifting (P,A) left one bit; else if $r < 0$ (and hence $r < -1/4$, since otherwise it would have been eliminated by the first condition), then compute $2r + b$ by shifting and then adding, else $r \ge 1/4$ and subtract b from $2r$. Using $b \ge 1/2$, it is easy to check that these rules keep $-1/2 \le r < 1/2$. For nonrestoring division, we only have $|r| \le b$, and we need P to be $n + 1$ bits wide. But for SRT division, the bound on r is tighter, namely $-1/2 \le r < 1/2$. Thus, we can save a bit by eliminating the high-order bit of P (and b and the adder). In particular, the test for equality of the top three bits of P becomes a test on just two bits.

P	A	
00000	1000	B contains 0011, so shift all registers left two places
00010	0000	B now contains 1100. Top bits of P are equal, so shift and set q_0= 0
00100	0000	Top bits are not equal, so set q_1= 1
01000	0000	shift and
+10100		subtract B
11100	0000	Top bits equal, so shift and set q_2= 0
11000	0000	Top bits are unequal, so set q_3 = –1
10000	0000	shift and
+01100		add B
11100		Remainder is negative, so restore it and subtract 1 from q_0
+01100		
01000		This must be shifted right two places to give remainder
		Remainder = 10,q = 010$\bar{1}$ –1 = 0010

FIGURE A.19 SRT division of 1000/0011.

The algorithm might change slightly in an implementation of SRT division. After each ALU operation, the P register can be shifted as many places as necessary to make either P ≥ 1/4 or P < −1/4. By shifting k places, k quotient bits are set equal to zero all at once. For this reason SRT division is sometimes described as one that keeps the remainder normalized to $|r| \geq 1/4$.

Notice that the value of the quotient bit computed in a given step is based on which operation is performed in that step (which in turn depends on the result of the operation from the previous step). This is in contrast to nonrestoring division, where the quotient bit computed in ith step depends on the result of the operation in the same step. This difference is reflected in the fact that when the final remainder is negative, the last quotient bit must be adjusted in SRT division, but not in nonrestoring division. However, the key fact about the quotient bits in SRT division is that they can include $\bar{1}$. Therefore the quotient bits can't be stored in the low-order bits of the A register; furthermore, the quotient must be converted to ordinary two's complement in a full adder. A common way to do this is to accumulate the positive quotient bits in one register and the negative quotient bits in another, and then subtract the two registers after all the bits are known. Because there is more than one way to write a number in terms of the digits −1, 0, 1, SRT division is said to use a *redundant* quotient representation.

The differences between SRT division and ordinary nonrestoring division can be summarized as follows:

1. ALU decision rule: In nonrestoring division, it is determined by the sign of P; in SRT, it is determined by the two most significant bits of P.

2. Quotient determination: In nonrestoring division, it is immediate from the signs of P; in SRT, it must be computed in a full n-bit adder.

3. Speed: SRT division will be faster on operands that produce zero quotient bits.

Speeding Up Multiplication with a Single Adder

As mentioned before, shifting-over techniques are not used much in current hardware. We now discuss some methods that are in more widespread use. Methods that increase the speed of multiplication can be divided into two classes: those that use a single adder and those that use multiple adders. Let's first discuss techniques that use a single adder.

In the discussion of addition we noted that, because of carry propagation, it is not practical to perform addition with two levels of logic. Using the cells of Figure A.13, adding two 64-bit numbers will require a trip through seven cells to compute the P's and G's, and seven more to compute the carry bits, which will require at least 28 logic levels. Each multiplication step will require a trip through this adder. A way to avoid this computation in each step is to use *carry-save adders* (CSA). A carry-save adder is simply n independent full adders. A

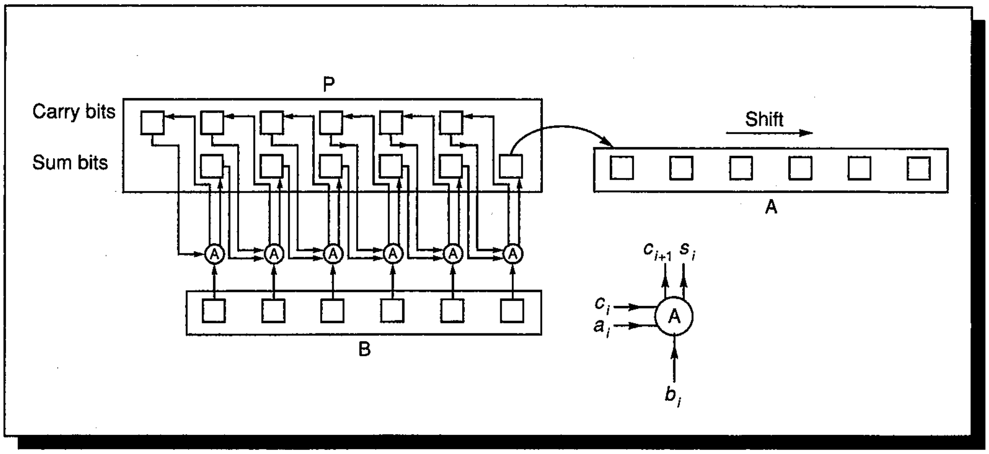


FIGURE A.20 Carry-save multiplier. Each circle represents a (3,2) adder working independently. At each step, the only bit of P that needs to be shifted is the low-order sum bit.

multiplier using such an adder is illustrated in Figure A.20. Each circle marked "A" is a single-bit full adder, and each box represents one bit of a register. Each addition operation results in a pair of bits, stored in the sum and carry parts of P. Since each add is independent, only two logic levels are involved in the add—a vast improvement over 28.

To operate the multiplier in Figure A.20, load the sum and carry bits of P with zero and perform the first ALU operation. (If Booth recoding is used, it might be a subtraction rather than an addition.) Then shift the low-order sum bit of P into A, as well as shifting A itself. The $n - 1$ high-order bits of P don't need to be shifted because on the next cycle the sum bits are fed into the next lower order adder. Each addition step is dramatically increased in speed, since each add cell is working independently of the others, and no carry is propagated. There are two drawbacks to carry-save adders. First, they require more hardware because there must be a copy of register P to hold the carry outputs of the adder. Second, after the last step, the high-order word of the result must be fed into an ordinary adder to combine the sum and carry parts. This could be accomplished by feeding the output of P into the adder used to perform the addition operation. Multiplying with a carry-save adder is sometimes called redundant multiplication because P is represented using two registers. Since there are many ways to represent P as the sum of two registers, this representation is redundant. The term *carry-propagate adder* (CPA) is used to denote an adder that is not a CSA. A propagate adder may propagate its carries using ripples, carry lookahead, or some other method.

Another way to speed up multiplication without using extra adders is to examine k low-order bits of A at each step, rather than just one bit. This is often called *higher-radix multiplication.* As an example, suppose that $k = 2$. If the pair of bits is 00, add 0 to P, and if it is 01, add B. If it is 10, simply shift b one bit left before adding it to P. Unfortunately, if the pair is 11, it appears we would have to compute $b + 2b$. But this can be avoided by using a higher-radix version of Booth recoding. Imagine A as a base 4 number: When the digit 3 appears, change it to $\bar{1}$ and add 1 to the next higher digit to compensate. The name for this technique, *overlapping triplets*, comes from the fact that it looks at 3 bits to determine what multiple of b to use, whereas ordinary Booth recoding looks at 2 bits.

The precise rules for overlapping triplets are given in Figure A.21. Besides having more complex control logic, this technique also requires that the P register be one bit wider to accommodate the possibility of $2b$ or $-2b$ being added to it. It is also possible to use a radix-8 (or even higher) version of Booth recoding. In that case, however, it will be necessary to use the multiple 3B as a potential summand. Radix-8 multipliers normally compute 3B once and for all at the beginning of a multiplication operation.

Current pair		**Previous**	**Multiple**
$i+1$	i	$i-1$	
0	0	0	0
0	0	1	$+b$
0	1	0	$+b$
0	1	1	$+2b$
1	0	0	$-2b$
1	0	1	$-b$
1	1	0	$-b$
1	1	1	0

FIGURE A.21 Multiples of *b* to use for radix-4 Booth recoding. For example, if the two low-order bits of the A register are both 1, and the last bit to be shifted out of the A register was 0, then the correct multiple is $-b$, obtained from the second to last row of the table.

Faster Multiplication with Many Adders

If the space for many adders is available, then multiplication speed can be improved. Figure A.22 shows a block diagram of a simple *array multiplier* for multiplying two 8-bit numbers using seven CSAs and one propagate adder. As it still takes eight additions to compute the product, the latency of computing a product is not dramatically different from using a single carry-save adder. However, with the hardware in Figure A.22, multiplication can be pipelined, increasing the total throughput. On the other hand, although this level of pipelining is sometimes used in array processors, it is not used in any of the single-chip, floating-point accelerators discussed in Section A.10. Pipelining is discussed in general in Chapter 6 and by Kogge [1981] in the context of multipliers.

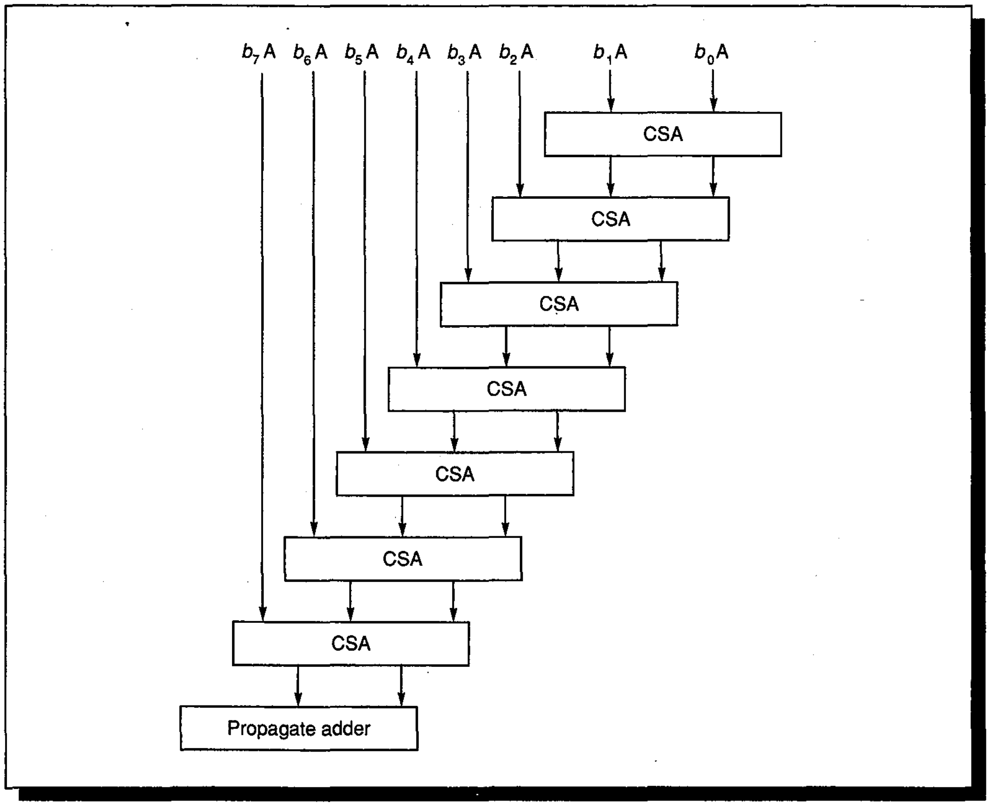


FIGURE A.22 Block diagram of an array multiplier. The 8-bit number in A is multiplied by $b_7 b_6 \cdots b_0$. Each box marked "CSA" is a carry-save adder.

With the technology of 1990, it is not possible to fit an array large enough to multiply two double-precision numbers on a single chip and have space left over for the other arithmetic operations. Thus, a popular design is to use a two-pass arrangement such as the one shown in Figure A.23 (page A-46). The first pass through the array "retires" four bits of B. Then the result of this first pass is fed back into the top to be combined with the next four summands. The result of this second pass is then fed into a CPA. This design, however, loses the ability to be pipelined.

If arrays require as many addition steps as the much cheaper arrangement in Figure A.2, why are they so popular? First of all, using an array has a smaller latency than using a single adder—because the array is a combinational circuit, the signals flow through it directly without being clocked. Although the two-pass adder of Figure A.23 would normally still use a clock, the cycle time for passing through k arrays can be less than k times the clock that would be needed for a design like the one in Figure A.2. Secondly, the array is amenable to various schemes for further speedup. One of them is shown in Figure A.24 (page A-47). The idea of this design is that two adds proceed in parallel or, to put it another way, each stream passes through only half the adders. Thus, it runs at almost twice the speed of the multiplier in Figure A.22. This *even/odd* multiplier is popular in VLSI because of its regular structure. Arrays can also be speeded up using asynchronous logic. One of the reasons why the multiplier of Figure A.2 (page A-4) needs a clock is to keep the output of the adder from feeding back into the input of the adder before the output has fully stabilized. Thus, if the array in Figure A.23 is long enough so that no signal can propagate from the top through the bottom in the time it takes for the first adder to stabilize, it may be possible to avoid clocks altogether. Williams et al. [1987] discusses a design using this idea, although it is for dividers instead of for multipliers.

The techniques of the previous paragraph still have a multiply time of O(n), but the time can be reduced to log n using a tree. The simplest tree would combine pairs of summands b_0A ··· b_{n-1}A, cutting the number of summands from n to n/2. Then these n/2 numbers would be added in pairs again, reducing to n/4, and so on, and resulting in a single sum after log n steps. However, this simple binary-tree idea doesn't map into full (3,2) adders, which reduce three inputs to two rather than reducing two inputs to one. A tree that does use full adders, known as a *Wallace tree*, is shown in Figure A.25. When computer arithmetic units were built out of MSI parts, a Wallace tree was the design of choice for high-speed multipliers. There is, however, a problem with implementing them in VLSI.

Figures A.22–A.24 are sufficiently concise that it may be hard to visualize all the adders involved in an array multiplier. Figure A.26 (page A-49) shows each individual adder in a 4-bit array multiplier. Figure A.26(b) shows the inputs to the circuit, and Figure A.26(c) shows how those inputs are connected by adders.

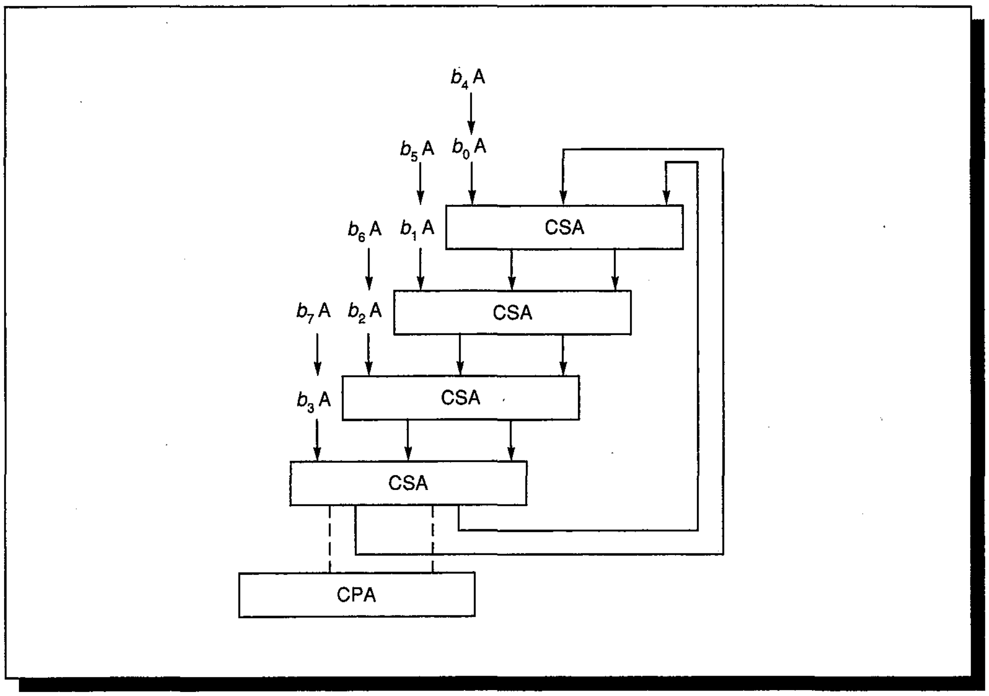


FIGURE A.23 Multipass array multiplier. Multiplies two 8-bit numbers with about half the hardware of that in Figure A.22. At the end of the second pass, the bits flow into the CPA.

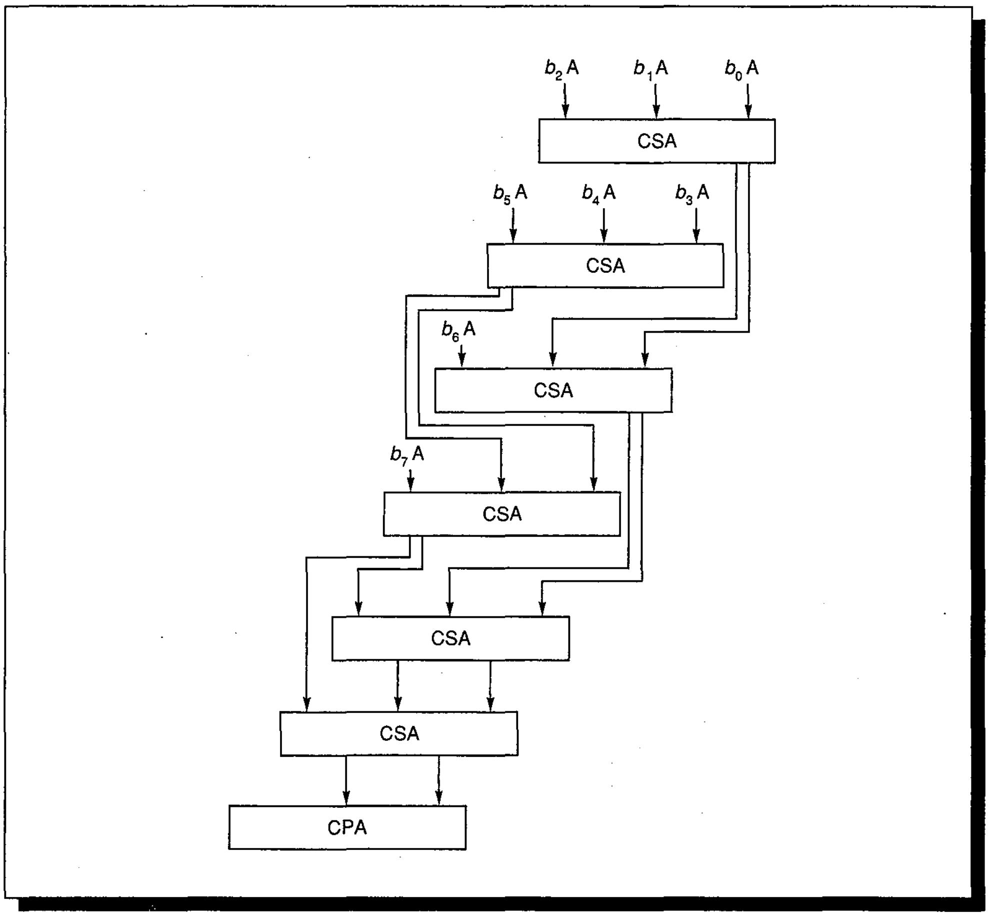


FIGURE A.24 Even/odd array. The first two adders work in parallel. Their results are fed into the third and fourth adders, which also work in parallel, and so on.

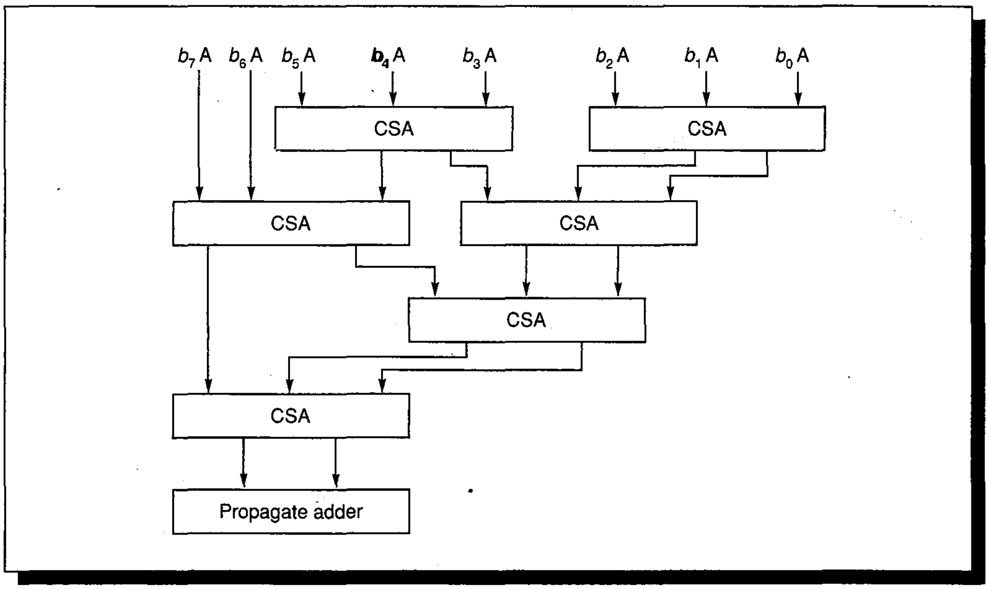


FIGURE A.25 Wallace-tree multiplier.

Each row of adders in A.26(c) corresponds to a single box in A.26(a). In actual implementation the array would be laid out as a square, not "twisted" as shown in the picture. (Lining up bits of the same significance in the same column makes the picture easier to understand.) If you try to fill in all the adders and paths for the Wallace tree of Figure A.25 (page A-47), you will discover that it does not have the nice, regular structure of Figure A.26. This is why VLSI designers have often chosen to use other log *n* designs such as the *binary-tree multiplier*, which is discussed next.

The problem with adding summands in a binary tree is that of coming up with a (2,1) adder that combines two digits and produces a single-sum digit. Because of carries, this isn't possible using binary notation, but it can be done with some other representation. We will use the *signed-digit representation* 1, $\overline{1}$, and 0, which we used previously to understand Booth's algorithm. This representation has two costs. First, it takes two bits to represent each signed digit. Second, the algorithm for adding two signed-digit numbers a_i and b_i is complex and requires examining $a_ia_{i-1}a_{i-2}$ and $b_ib_{i-1}b_{i-2}$. Although this means you must look two bits back, in binary addition you might have to look an arbitrary number of bits back (because of carries).

We can describe the algorithm for adding two signed-digit numbers as follows. First, compute sum and carry bits s_i and c_{i+1} using the table in Figure A.27. Then compute the final sum as $s_i + c_i$. The tables are set up so that this final sum does not generate a carry.

Example: What is the sum of the signed-digit numbers $1\overline{1}0$ and 001 ?

Answer: The two low-order bits sum to $0 + 1 = 1\overline{1}$, the next pair sums to $\overline{1} + 0 = 0\overline{1}$, and the high-order pair sums to $1 + 0 = 01$, so the sum is $1\overline{1} + 0\overline{1}0 + 0100 = 10\overline{1}$.

This, then, defines a (2,1) adder. With this in hand, we can use a straightforward binary tree to perform multiplication. In the first step it adds b_0A + b_1A in parallel with b_2A + b_3A, ···, b_{n-2}A + b_{n-1}A. The next step adds the results of these sums in pairs, and so on. Although the final sum must be run through a carry-propagate adder to convert it from signed-digit form to two's complement, this final add step is necessary in any multiplier using CSAs.

To summarize, both Wallace trees and signed-digit trees are log *n* multipliers. The Wallace tree uses the fewer gates but is harder to lay out. The signed-digit tree has a more regular structure, but requires two bits to represent each digit and has more complicated add logic. As with adders, it is possible to combine different multiply techniques. For example, Booth recoding and arrays can be combined. In Figure A.22 (page A-45) instead of having each input be b_iA, we could have it be b_ib_{i-1}A, and in order to avoid having to compute the multiple 3*b*, we can use Booth recoding.

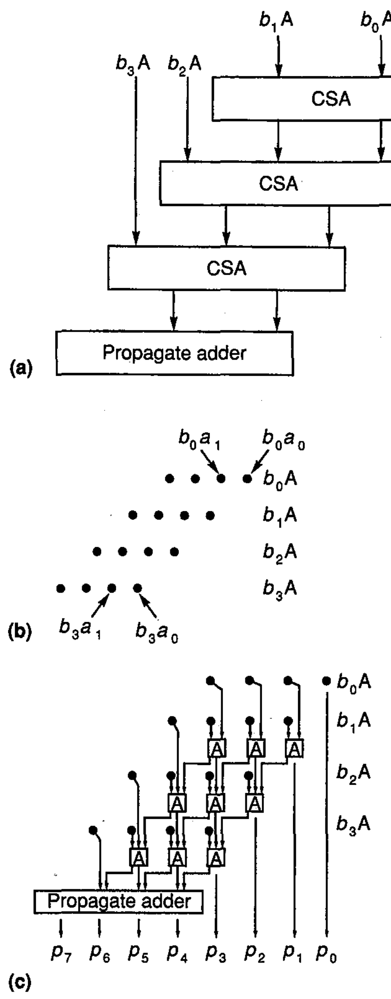


FIGURE A.26 Block diagram of an array multiplier (a); the inputs to the array (b); the array expanded to show all the adders (c).

$$
\begin{array}{cccccc}
\begin{array}{r}1\\ \underline{+1}\\ 1\,0\end{array} &
\begin{array}{r}1\\ \underline{+\bar{1}}\\ 0\,0\end{array} &
\begin{array}{r}\bar{1}\\ \underline{+\bar{1}}\\ \bar{1}\,0\end{array} &
\begin{array}{r}0\\ \underline{+0}\\ 0\,0\end{array} &
\begin{array}{ll}1\ x & \\ \underline{+0\ y} & \\ 1\,\bar{1} & \text{if } x \geq 0 \text{ and } y \geq 0\\ 0\,1 & \text{otherwise}\end{array} &
\begin{array}{ll}\bar{1}\ x & \\ \underline{+0\ y} & \\ 0\,\bar{1} & \text{if } x \geq 0 \text{ and } y \geq 0\\ \bar{1}\,1 & \text{otherwise}\end{array}
\end{array}
$$

FIGURE A.27 Signed-digit addition table. The leftmost sum shows that when computing 1 + 1, the sum bit is 0 and the carry bit is 1.

Faster Division with One Adder

The two techniques for speeding up multiplication with a single adder were carry-save adders and higher-radix multiplication. There is a difficulty when trying to utilize these approaches to speed up nonrestoring division. The problem with CSAs is that at the end of each cycle the value of P, since it is in carry-save form, is not known exactly. In particular, the sign of P is uncertain, yet it is the sign of P that is used to compute the quotient digit and decide on the next ALU operation. When a higher radix is used, the problem is deciding what value to subtract from P. In the paper-and-pencil method, you have to guess the quotient digit. In binary division there are only two possibilities; we were able to finesse the problem by initially guessing one and then adjusting the guess based on the sign of P. This doesn't work in higher radices because there are more than two possible quotient digits, rendering quotient selection potentially quite complicated: You would have to compute all the multiples of *b* and compare them to P.

Both the carry-save technique and higher-radix division can be made to work if we use a redundant quotient representation. Recall from our discussion of SRT division that by allowing the quotient digits to be −1, 0, or 1, there is often a choice of which one to pick. The idea in the previous algorithm was to choose zero whenever possible because that meant an ALU operation could be skipped. In carry-save division, the idea is that because the remainder (P register) is not known exactly (being stored in carry-save form), the exact quotient digit is also not known. But thanks to the redundant representation, the remainder doesn't have to be known precisely in order to pick a quotient digit. This is illustrated in Figure A.28, where the *x* axis represents r_i, the contents of the (P,A) register pair after *i* steps. The line labeled $q_i = 1$ shows the value that r_{i+1} would be if we choose $q_i = 1$, and similarly for the lines $q_i = 0$ and $q_i = -1$. We can choose any value for q_i, as long as $r_{i+1} = r\text{P}_i - q_i\text{B}$ satisfies $|r_{i+1}| \leq \text{B}$. The allowable ranges are shown in the right half of Figure A.28. Thus we only need to know *r* precisely enough to decide in which range in Figure A.28 it lies.

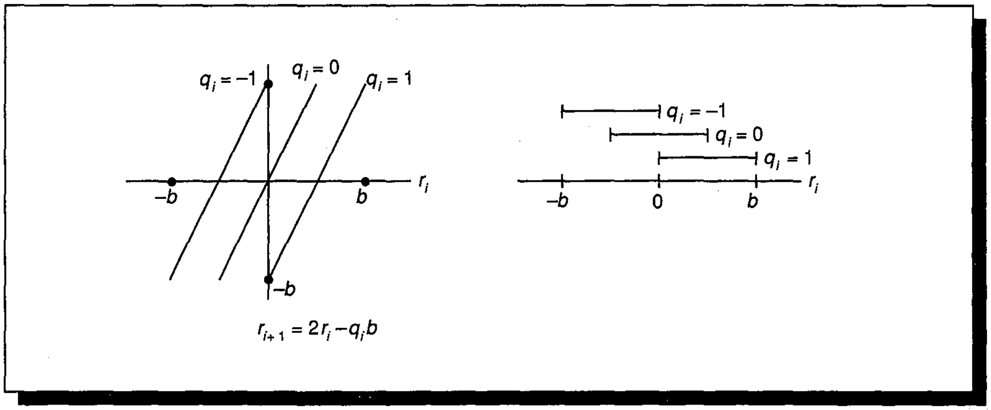


FIGURE A.28 Quotient selection for radix-2 division. The *x* axis represents the *i* th remainder, which is the quantity in the (P,A) register pair. The *y* axis shows the value of the remainder after one additional divide step. Each bar on the right-hand graph gives the range of r_i values for which it is permissible to select the associated value of q_i.

This is the basis for using carry-save adders. Look at the high-order bits of the carry-save adder and sum them in a propagate adder. Then use this approximation of r to compute q_i, usually by means of a lookup table. The same technique works for higher-radix division (whether or not a carry-save adder is used). The high-order bits P can be used to index a table that gives one of the allowable quotient digits.

The design challenge when building a high-speed SRT divider is figuring out how many bits of P and B need to be examined. For example, suppose that we take a radix of 4, use quotient digits of 2, 1, 0,$\bar{1}$, $\bar{2}$, but have a propagate adder. How many bits of P and B need to be examined? Deciding this involves two steps. For ordinary radix-2 nonrestoring division, because at each stage $|r| \leq b$, the P buffer won't overflow. But for radix 4, $r_{i+1} = 4r_i - q_i b$ is computed at each stage, and if r_i is near b, then $4r_i$ will be near $4b$, and even the largest quotient digit will not bring r back to the range $|r_{i+1}| \leq b$. In other words, the remainder might grow without bound. However, restricting $|r_i| \leq 2b/3$ makes it easy to check that r_i will stay bounded.

After figuring out the bound that r_i must satisfy, we can draw the diagram in Figure A.29, which is analogous to Figure A.28. If r_i is between $(1/12)b$ and $(5/12)b$, we can pick $q = 1$, and so on. Or to put it another way, if r/b is between 1/12 and 5/12, we can pick $q = 1$. Suppose we look at 4 bits of P and 4 bits of b, and the high bits of P (not counting the $(n + 1)$-st sign bit) are $0011xxx\cdots$, while the high bits of b are $1001xxx\cdots$. To simplify calculation, imagine the binary point at the left end of each register. Since we truncated, r (the value of P concatenated with A) could have a value from .0011 to .0100, and b could have a value from .1001 to .1010. Thus r/b could be as small as .0011/.1010 or as large as .0100/.1001. But $.0011_2/.1010_2 = 3/10 < 1/3$ would require a quotient bit of 1, while $.0100_2/.1001_2 = 4/9 > 5/12$ would require a quotient bit of 2. In other words, 4 bits of P and 4 bits of b aren't enough to pick a quotient bit. It turns out that 5 bits of P and 4 bits of b are enough. This can be verified by writing a simple program that checks all the cases.

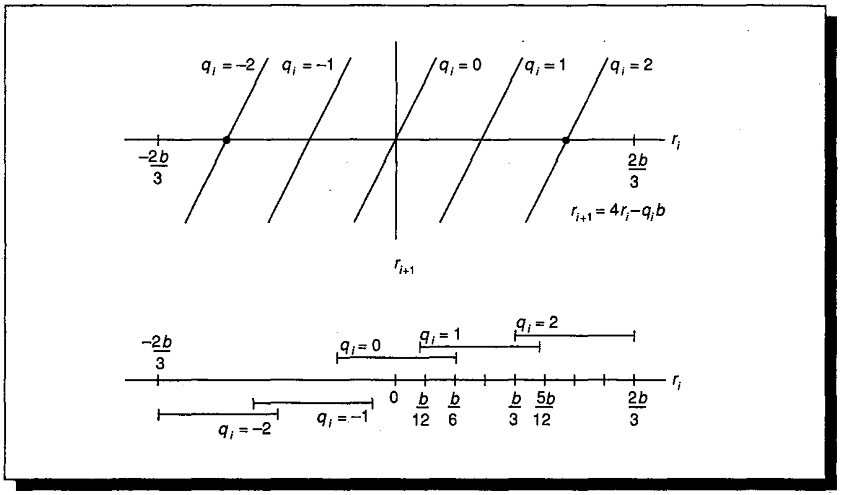


FIGURE A.29 Quotient selection for radix-4 division.

Example: Suppose that the radix is 4 and the quotient digits are 2, 1, $0, \bar{1}, \bar{2}$, but this time a CSA is used instead of a propagate adder. How many bits of the P and B registers need to be examined?

Answer: Once again $|r_i| \leq 2b/3$, and the ranges of the q_i are still as in Figure A.29. If the top 4 bits of the sum part and the carry part of P are respectively 0010 and 0001, then the sum part ranges from 0010 to 0011 and the carry part from 0001 to 0010. Accordingly, the true value of r ranges from 0010 + 0001 = 0011 to 0011 + 0010 = 0101. Given, therefore, a CPA that adds the top 4 bits of the carry and sum parts of P, and a sum of 0011, the true sum will be anywhere from 0011 to 0101. A program that checks all the cases will show that 6 bits of P and 4 bits of b are needed to predict a quotient digit. The result of such a program is shown in Figure A.30. For example, if b is 1001*xxx*··· and r is 001101*xxx*···, then the top 4 bits of b are 9 and the top 6 bits of r are 13, making the quotient digit 1. But if r were 001110_2 = 14, the quotient digit would have to be 2.

b	Range of P		*q*	*b*	Range of P		*q*
8	–21	–14	–2	12	–32	–20	–2
8	–13	–5	–1	12	–20	–7	–1
8	–5	3	0	12	–8	6	0
8	3	11	1	12	5	18	1
8	12	21	2	12	18	32	2
9	–24	–16	–2	13	–34	–21	–2
9	–15	–6	–1	13	–21	–7	–1
9	–6	4	0	13	–8	6	0
9	4	13	1	13	5	19	1
9	14	24	2	13	19	34	2
10	–26	–17	–2	14	–37	–22	–2
10	–16	–6	–1	14	–23	–7	–1
10	–6	4	0	14	–9	7	0
10	4	14	1	14	5	21	2
10	15	26	2	14	20	37	2
11	–29	–18	–2	15	–40	–24	–2
11	–18	–6	–1	15	–25	–8	–1
11	–7	5	0	15	–10	8	0
11	4	16	1	15	6	23	1
11	16	29	2	15	22	40	2

FIGURE A.30 Quotient digits for radix-4 SRT division with a CSA. The top row says that if the high-order 4 bits of b are 1000_2 = 8, and if the top 6 bits of P are between 110010_2 = –14 and 101011_2 = –21, then the quotient digit is –2.

Although these are simple cases, all SRT analyses proceed in the same way. First compute the range of r_i, then plot r_i against r_{i+1} to find the quotient ranges, and finally write a program to compute how many bits are necessary. (It is sometimes also possible to compute the required number of bits analytically.) Two final comments about high-radix SRT division are in order. First, Figure A.30 is not symmetrical. Thus, for a radix-4 CSA divider, the lookup table needs not only 6 bits of P, but also the sign of P. Second, the quotient lookup table has a fairly regular structure. This means it is usually cheaper to encode it as a PLA rather than in ROM.

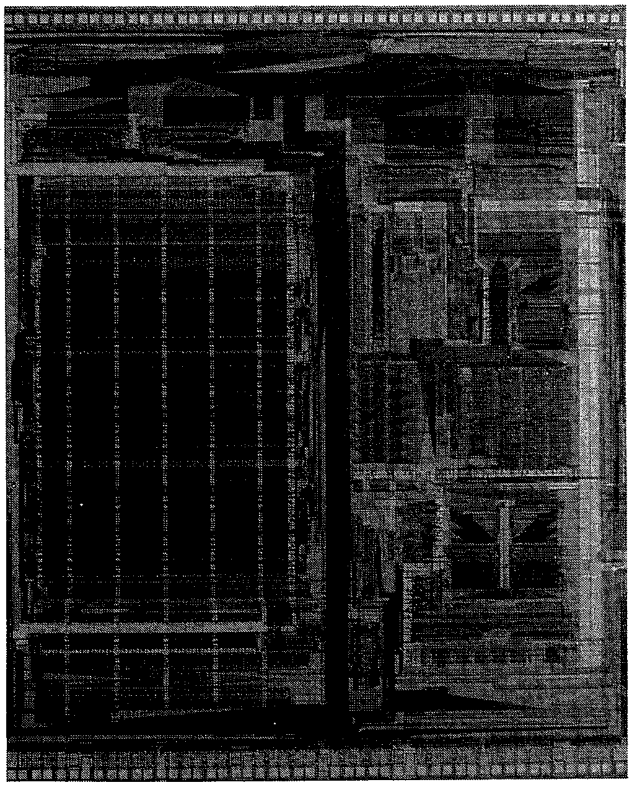
A.10 Putting It All Together

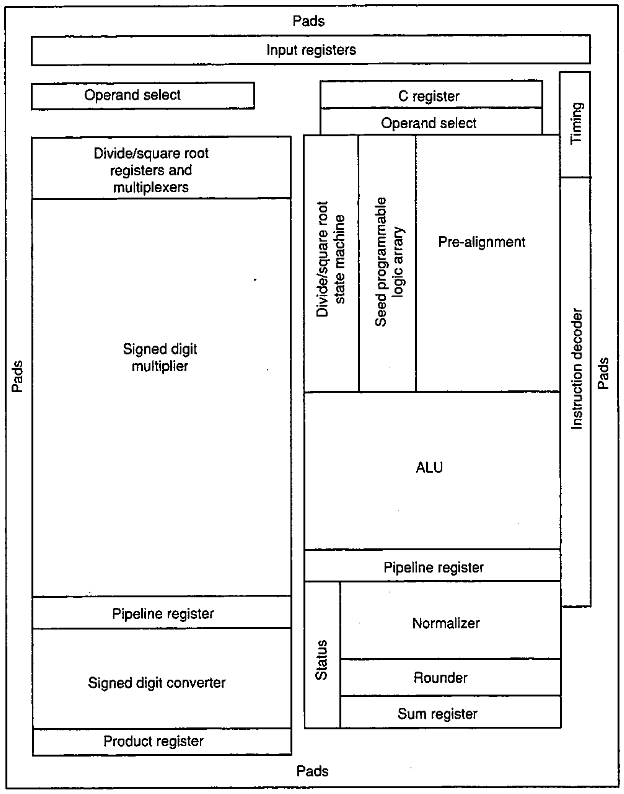
In this section, we will compare the Weitek 3364, the MIPS R3010, and the Texas Instruments 8847 (see Figures A.31 and A.32, pages A-54–A-55). In many ways, these are ideal chips to compare. They each implement the IEEE standard for addition, subtraction, multiplication, and division on a single chip. All were introduced in 1988 and run with a cycle time of about 40 nanoseconds. However, as we will see, they use quite different algorithms. The Weitek chip is well described in Birman et al. [1988], the MIPS chip is described in less detail in Rowen, Johnson, and Ries [1988], and the details of the TI chip have yet to be published.

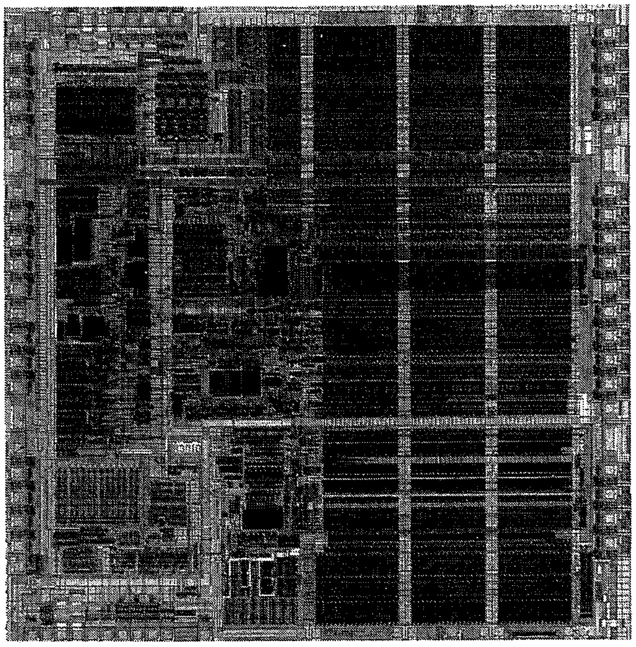
There are a number of things that these three chips have in common. They perform addition and multiplication in parallel, and they implement neither extended precision nor the IEEE remainder operation. We discussed earlier how an efficient REM could be provided in software if only chips would implement a remainder-step function. The designers of these chips probably decided not to

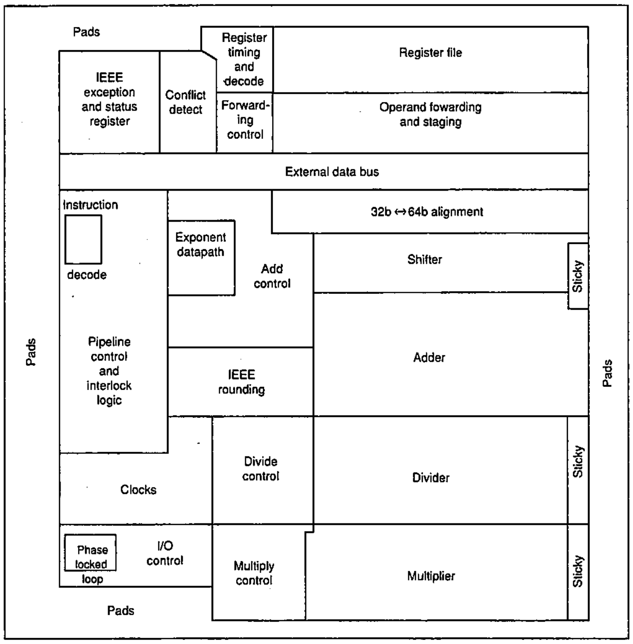
	MIPS R3010	Weitek 3364	TI 8847
Clock cycle time (ns)	40	50	30
Size (mil^2)	114,857	147,600	156,180
Transistors	75,000	165,000	180,000
Pins	84	168	207
Power (watts)	3.5	1.5	1.5
Cycles/add	2	2	2
Cycles/mult	5	2	3
Cycles/divide	19	17	11
Cycles/sq root	–	30	14

FIGURE A.31 Summary of the three floating-point chips discussed in this section. The cycle times are for production parts available in June 1989. The cycle counts are for double-precision operations.




Pads
Input registers
Operand select
C register
Operand select
Timing
Divide/square root registers and multiplexers
Divide/square root state machine
Seed programmable logic arrary
Pre-alignment
Signed digit multiplier
Pads
Instruction decoder
Pads
ALU
Pipeline register
Pipeline register
Normalizer
Status
Signed digit converter
Rounder
Sum register
Product register
Pads




Pads
Register timing and decode
Register file
IEEE exception and status register
Conflict detect
Forward-ing control
Operand fowarding and staging
External data bus
Instruction
decode
32b ↔ 64b alignment
Exponent datapath
Add control
Shifter
Sticky
Pads
Pipeline control and interlock logic
Adder
Pads
IEEE rounding
Divide control
Divider
Sticky
Clocks
Phase locked loop
I/O control
Multiply control
Multiplier
Sticky
Pads

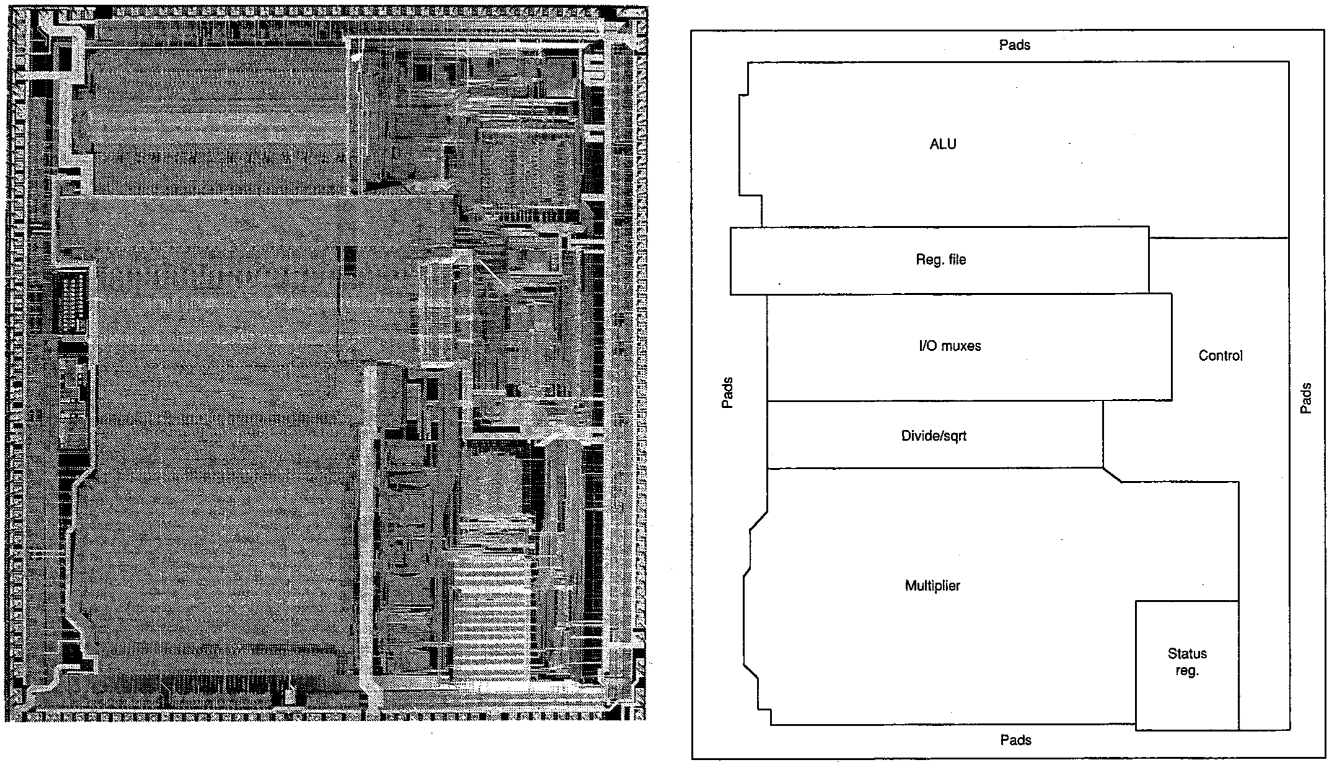


FIGURE A.32 Chip layout. In the left-hand column are the photomicrographs; the right-hand column shows the corresponding floor plans. Top left is the TI 8847, bottom left is the MIPS R3010, and above is the Weitek 3364.

provide extended precision because the most influential users are those who run portable codes, which can't rely on extended precision. However, as we have seen, extended precision can make for faster and simpler math libraries.

A summary of the three chips is given in Figures A.31 (page A-53) and A.32. Note that a higher transistor count generally leads to smaller cycle counts. Comparing the cycles/op numbers needs to be done carefully because the figures for the MIPS chip are those for a complete system (R3000/3010 pair), while the Weitek and TI numbers are for standalone chips, and are usually larger when used in a complete system.

The MIPS chip has the fewest transistors of the three. This is reflected in the fact that it is the only chip of the three that does not have any pipelining or hardware square root. Further, the multiplication and addition operations are not completely independent because they share the carry-propagate adder that performs the final rounding (as well as the rounding logic). Addition on the R3010 uses a mixture of ripple, CLA, and carry select. A carry-select adder is used in the fashion of Figure A.16 (page A-38). Within each half, carries are propagated using a hybrid ripple-CLA scheme of the type indicated in Figure A.14. However, this is further tuned by varying the size of each block, rather than having each fixed at four bits (as they are in Figure A.14 on page A-36). The multiplier is midway between the designs of Figures A.2 (page A-4) and A.22 (page A-45). It has an array just large enough so that output can be fed back into the input without having to be clocked. Also, it uses radix-4 Booth recoding and the even-odd technique of Figure A.24 (page A-47). The R3010 can do a divide and multiply in parallel (like the Weitek chip but unlike the TI chip). The divider is a radix-4 SRT method with quotient digits −2, −1, 0, 1, and 2, and is similar to that described in Taylor [1985]. Double-precision division is about four times slower than multiplication. The R3010 shows that for chips using an O(n) multiplier, an SRT divider can operate fast enough to keep a reasonable ratio between multiply and divide.

The Weitek 3364 has independent add, multiply, and divide units, and also uses radix-4 SRT division. However, the add and multiply operations on the Weitek chip are pipelined. The three addition stages are (1) exponent compare, (2) add followed by shift (or vice versa), and (3) final rounding. Stages (1) and (3) take only a half-cycle, allowing the whole operation to be done in two cycles, even though there are three pipline stages. The multiplier uses an array of the style of Figure A.23 but uses radix-8 Booth recoding, which means it must compute 3 times the multiplier. The three multiplier pipeline stages are (1) compute 3b, (2) pass through array, and (3) final carry-propagation add and round. Single precision passes through the array once, double precision twice. Like addition, the latency is two cycles. The Weitek chip uses an interesting addition algorithm. It is a variant on the carry-skip adder pictured in Figure A.15 (page A-37). However P_{ij}, which is the logical AND of many terms, is computed by rippling, performing one AND per ripple. Thus, while the carries propagate left within a block, the value of P_{ij} is propagating right within the next block, and the block sizes are chosen so that both waves complete at the same time. Unlike the MIPS chip, the 3364 has hardware square root, which shares the divide hardware. The ratio of double-precision multiply to divide is 2:17. The large disparity between multiply and divide is due to the fact that multiplication uses radix-8 Booth recoding, while division uses a radix-4 method. In the MIPS R3010, multiplication and division use the same radix.

The notable feature of the TI 8847 is that it does division by iteration (using the Goldschmidt algorithm discussed in Section A.6). This improves the speed of division (the ratio of multiply to divide is 3:11), but means that multiplication and division cannot be done in parallel as on the other two chips. Addition has a two-stage pipeline. Exponent compare, fraction shift, and fraction addition are done in the first stage, normalization and rounding in the second stage. Multiplication uses a binary tree of signed-digit adders and has a three-stage pipeline. The first stage passes through the array retiring half the bits, the second stage passes through the array a second time, and the third stage converts from signed-digit form to two's complement. Since there is only one array, a new multiply operation can only be initiated in every other cycle. However, by slowing down the clock, two passes through the array can be made in a single cycle. In this case, a new multiplication can be initiated in each cycle. The 8847 adder uses a carry-select algorithm rather than carry lookahead. As mentioned in Section A.6, the TI carries 60 bits of precision in order to do correctly rounded division.

These three chips illustrate the different tradeoffs made by designers with similar constraints. One of the most interesting things about these chips is the diversity of their algorithms. Each uses a different add algorithm, as well as a different multiply algorithm. In fact, Booth recoding is the only technique that is universally used by all the chips.

A.11 Fallacies and Pitfalls

Fallacy: Underflows rarely occur in actual floating-point application code.

Although most codes rarely underflow, there are actual codes that underflow frequently. SDRWAVE [Kahaner 1988], which solves a one-dimensional wave equation, is one such example. This program underflows quite frequently, even when functioning properly. Measurements on one machine show that adding hardware support for gradual underflow would cause SDRWAVE to run about 50% faster.

Fallacy: Conversions between integer and floating point are rare.

In fact, in Spice they are as frequent as divides. The assumption that conversions are rare leads to a mistake in the SPARC instruction set, which does not provide an instruction to move from integer registers to floating-point registers.

Pitfall: Don't increase the speed of a floating-point unit without increasing its memory bandwidth.

A typical use of a floating-point unit is to add two vectors to produce a third vector. If these vectors consist of double-precision numbers, then each floating-point add will use three operands of 64 bits each, or 24 bytes of memory. The memory bandwidth requirements are even greater if the floating-point unit can perform addition and multiplication in parallel (as most do).

Pitfall: –x is not the same as 0 – x.

This is a fine point in the IEEE standard that has tripped up some designers. Because floating-point numbers use the sign/magnitude system, there are two zeros, +0 and –0. The standard says that $0 - 0 = +0$, whereas $-(0) = -0$. Thus $-x$ is not the same as $0 - x$ when $x = 0$.

A.12 Historical Perspective and References

The earliest computers used fixed point rather than floating point. In "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument," Burks, Goldstine, and von Neumann put it like this:

> *There appear to be two major purposes in a "floating" decimal point system both of which arise from the fact that the number of digits in a word is a constant fixed by design considerations for each particular machine. The first of these purposes is to retain in a sum or product as many significant digits as possible and the second of these is to free the human operator from the burden of estimating and inserting into a problem "scale factors" — multiplicative constants which serve to keep numbers within the limits of the machine.*
>
> *There is, of course, no denying the fact that human time is consumed in arranging for the introduction of suitable scale factors. We only argue that the time so consumed is a very small percentage of the total time we will spend in preparing an interesting problem for our machine. The first advantage of the floating point is, we feel, somewhat illusory. In order to have such a floating point, one must waste memory capacity which could otherwise be used for carrying more digits per word. It would therefore seem to us not at all clear whether the modest advantages of a floating binary point offset the loss of memory capacity and the increased complexity of the arithmetic and control circuits.* [Bell and Newell 1971, 97]

This enables us to see things from the perspective of early computer designers, who believed that saving computer time and memory were more important than saving programmer time.

The original papers introducing the Wallace tree, Booth recoding, SRT division, overlapped triplets, and so on, are reprinted in Swartzlander [1980]. A good explanation of an early machine (the IBM 360/91) that used a pipelined Wallace tree, Booth recoding, and iterative division is in Anderson et al. [1967]. A discussion of the average time for single-bit SRT division is in Freiman [1961]; this is one of the few interesting historical papers that does not appear in Swartzlander.

The standard book of Mead and Conway [1980] discouraged the use of CLAs as not being cost effective in VLSI. Brent and Kung [1982] was an important paper that helped combat that view. An example of a detailed layout for CLAs can be found in Ngai and Irwin [1985] or in Weste and Eshraghian [1985]. Takagi, Yasuura, and Yajima [1985] provides a detailed description of a signed-digit–tree multiplier.

Although the IEEE standard is being widely adopted, there are still three other important floating-point systems in use: the IBM/370, the DEC VAX, and the Cray. We will briefly discuss these older formats. The VAX format is closest to the IEEE standard. Its single-precision format (F format) is like IEEE single precision in that it has a hidden bit, 8 bits of exponent, and 23 bits of fraction. However, it does not have a sticky bit, which causes it to round halfway cases up instead of to even. The VAX has a slightly different exponent range than IEEE single: $E_{\min}$ is -128 rather than -126 as in IEEE, and $E_{\max}$ is 126 instead of 127. The main differences between VAX and IEEE are the lack of special values and gradual underflow. The VAX has a reserved operand, but it works like a signaling NaN: it traps whenever it is referenced. Originally, the VAX's double precision (D format) also had 8 bits of exponent. However, as this is too small for many applications, a G format was added; like the IEEE standard, this format has 11 bits of exponent. The VAX also has an H format, which is 128 bits long.

The IBM/370 floating-point format uses base 16 rather than base 2. This means it cannot use a hidden bit. In single precision, it has 7 bits of exponent and 24 bits (6 hex digits) of fraction. Thus, the largest representable number is $16^{2^7} = 2^{4 \times 2^7} = 2^{2^9}$, compared with 2^{2^8} for IEEE. However, a number that is normalized in the hexadecimal sense only needs to have a nonzero leading digit. When interpreted in binary, the three most significant bits could be zero. Thus, there are potentially fewer than 24 bits of significance. The reason for using the higher base was to minimize the amount of shifting required when adding floating-point numbers. However, this is less significant in current machines, where the floating-point add time is usually fixed independent of the operands. Another difference between 370 arithmetic and IEEE arithmetic is that the 370 has neither a round digit nor a sticky digit, which effectively means that it truncates rather than rounds. Thus, in many computations, the result will systematically be too small. Unlike the VAX and IEEE arithmetic, every bit pattern is a valid number. Thus, library routines must establish conventions for what to return in case of errors. In the IBM FORTRAN library, for example, $\sqrt{-4}$ returns 2!

Arithmetic on Cray computers is interesting because it is driven by a motivation for the highest possible floating-point performance. It has a 15-bit exponent field and a 48-bit fraction field. Addition on Cray computers does not have a guard digit, and multiplication is even less accurate than addition. Thinking of multiplication as a sum of p numbers, each $2p$ bits long, what Cray computers do is to drop the low-order bits of each summand. Thus, analyzing the exact error characteristics of the multiply operation is not easy. Reciprocals are computed using iteration, and division of a by b is done by multiplying a times $1/b$. The errors in multiplication and reciprocation combine to make the last three bits of a divide operation unreliable. At least Cray computers serve to keep numerical analysts on their toes!

The IEEE standardization process began in 1977, inspired mainly by W. Kahan, and is based partly on Kahan's work with the IBM 7094 at the University of Toronto [Kahan 1968]. The standardization process was a lengthy affair, with gradual underflow causing the most controversy. (According to Cleve Moler, visitors to the U.S. were advised that the sights not to be missed were Las Vegas, the Grand Canyon, and the IEEE standards committee meeting.) The standard was finally approved in 1985. The Intel 8087 was the first major commercial IEEE implementation and appeared in 1981, before the standard was finalized. It contains features that were eliminated in the final standard, such as projective bits. According to Kahan, the length of double-extended precision was based on what could be implemented in the 8087. Although the IEEE standard was not based on any existing floating-point system, most of its features were present in some other system. For example the CDC 6600 reserved special bit patterns for INDEFINITE and INFINITY, while the idea of denormal numbers appears in Goldberg [1967] as well as in Kahan [1968]. Kahan was awarded the 1989 Turing prize in recognition of his work on floating point.

References

ANDERSON, S. F., J. G. EARLE, R. E. GOLDSCHMIDT, AND D. M. POWERS [1967]. "The IBM System/360 Model 91: Floating-point execution unit," *IBM J. Research and Development* 11, 34–53. Reprinted in [Swartzlander 1980].

Good description of an early high-performance floating-point unit that used a pipelined Wallace-tree multiplier and iterative division.

ATKINS, D. E. [1968]. "Higher-radix division using estimates of the divisor and partial remainders," *IEEE Trans. on Computers* C-17:10, 925–934. Reprinted in [Swartzlander 1980].

This is the standard reference for high-radix SRT division.

BELL, C. G. AND A. NEWELL, [1971]. *Computer Structures: Readings and Examples,* McGraw-Hill, New York.

BIRMAN, M., G. CHU, L. HU, J. MCLEOD, N. BEDARD, F. WARE, L. TORBAN, AND C. M. LIM [1988]. "Design of a high-speed arithmetic datapath," *Proc. ICCD: VLSI Computers and Processors,* 214–216.

Fairly detailed description of the Weitek 3364 floating-point chip.

BRENT, R. P. AND H. T. KUNG [1982] "A regular layout for parallel adders," *IEEE Trans. on Computers* C-31, 260–264.

This is the paper that popularized CLA adders in VLSI.

BURKS, A. W., H. H. GOLDSTINE, AND J. VON NEUMANN, [1946]. *Preliminary Discussion of the Logical Design of an Electronic Computing Instrument.*

CODY, W. J. [1988]. "Floating point standards: Theory and practice," in *Reliability in Computing: The Role of Interval Methods in Scientific Computing,* R. E. Moore, (ed.), Academic Press, Boston, Mass., 99–107.

Presents a status of hardware and software implementations of the standard.

CODY, W. J., J. T. COONEN, D. M. GAY, K. HANSON, D. HOUGH, W. KAHAN, R. KARPINSKI, J. PALMER, F. N. RIS, AND D. STEVENSON [1984]. "A proposed radix- and word-length-independent standard for floating-point arithmetic," *IEEE Micro* 4:4, 86–100.

Contains a draft of the 854 standard, which is more general than 754. The significance of this article is that it contains commentary on the standard, most of which is equally relevant to 754.

COONEN, J. [1984]. *Contributions to a Proposed Standard for Binary Floating-Point Arithmetic,* Ph.D. Thesis, Univ. of Calif., Berkeley.

The only detailed discussion of how rounding modes can be used to implement efficient binary decimal conversion.

FREIMAN, C. V. [1961]. "Statistical analysis of certain binary division algorithms," *Proc. IRE* 49:1, 91–103.

Contains an analysis of the performance of shifting-over-zeros SRT division algorithm.

GOLDBERG, D. [1989]. "Floating-point and computer systems," *Xerox Tech. Rep.* CSL-89-9. A version of this paper will appear in *Computing Surveys.*

Contains an in-depth tutorial on the IEEE standard from the software point of view.

GOLDBERG, I. B. [1967]. "27 bits are not enough for 8-digit accuracy," *Comm. ACM* 10:2, 105–106.

This paper proposes using hidden bits and gradual underflow.

GOSLING, J. B. [1980]. *Design of Arithmetic Units for Digital Computers,* Springer-Verlag NewYork, Inc., New York.

A concise, well-written book, although it focuses on MSI designs.

HAMACHER, V. C., Z. G. VRANESIC, AND S. G. ZAKY [1984]. *Computer Organization,* 2nd ed., McGraw-Hill, New York.

Introductory computer architecture book with a good chapter on computer arithmetic.

HWANG, K. [1979]. *Computer Arithmetic: Principles, Architecture, and Design,* Wiley, New York.

This book contains the widest range of topics of the computer arithmetic books.

IEEE [1985]. "IEEE standard for binary floating-point arithmetic," *SIGPLAN Notices* 22:2, 9–25.

IEEE 754 is reprinted here.

KAHAN, W. [1968]. "7094-II system support for numerical analysis," *SHARE Secretarial Distribution* SSD-159.

This system had many features that were incorporated into the IEEE floating-point standard.

KAHANER, D. K. [1988]. "Benchmarks for 'real' programs," *SIAM News* (November).

The benchmark presented in this article turns out to cause many underflows.

KNUTH, D. [1981]. *The Art of Computer Programming,* vol II, 2nd ed., Addison-Wesley, Reading, Mass.

Has a section on the distribution of floating-point numbers.

KOGGE, P. [1981]. *The Architecture of Pipelined Computers,* McGraw-Hill, New York.

Has brief discussion of pipelined multipliers.

KOHN, L. AND S.-W. FU, [1989]. "A 1,000,000 transistor microprocessor," *IEEE Int'l Solid-State Circuits Conf.*, 54–55.

A brief overview of the Intel 860, whose floating-point addition algorithm is discussed in Section A.4.

MAGENHEIMER, D. J., L. PETERS, K. W. PETTIS, AND D. ZURAS, [1988]. "Integer multiplication and division on the HP Precision Architecture," *IEEE Trans. on Computers* 37:8, 980–990.

Rationale for the integer- and divide-step instructions in the Precision architecture.

MEAD, C. AND L. CONWAY [1980]. *Introduction to VLSI Systems,* Addison-Wesley, Reading, Mass.

NGAI, T-F. AND M. J. IRWIN [1985]. "Regular, area-time efficient carry-lookahead adders," *Proc. Seventh IEEE Symposium on Computer Arithmetic,* 9–15.

Describes a CLA adder like that of Figure A.13, where the bits flow up and then come back down.

PENG, V., S. SAMUDRALA, AND M. GAVRIELOV [1987]. "On the implementation of shifters, multipliers, and dividers in VLSI floating point units," *Proc. Eighth IEEE Symposium on Computer Arithmetic,* 95–102.

Highly recommended survey of different techniques actually used in VLSI designs.

ROWEN, C., M. JOHNSON, and P. RIES [1988]. "The MIPS R3010 floating-point coprocessor," *IEEE Micro* 53–62 (June).

SANTORO, M. R., G. BEWICK, and M. A. HOROWITZ [1989]. "Rounding algorithms for IEEE multipliers," *Proc. Ninth IEEE Symposium on Computer Arithmetic,* 176–183.

A very readable discussion of how to efficiently implement rounding for floating-point multiplication.

SCOTT, N. R. [1985]. *Computer Number Systems and Arithmetic,* Prentice-Hall, Englewood Cliffs, N.J.

SWARTZLANDER, E., ED. [1980]. *Computer Arithmetic,* Dowden, Hutchison and Ross (distributed by Van Nostrand, New York).

A collection of historical papers.

TAKAGI, N., H. YASUURA, AND S. YAJIMA [1985]."High-speed VLSI multiplication algorithm with a redundant binary addition tree," *IEEE Trans. on Computers* C-34:9, 789–796.

A discussion of the binary-tree signed multiplier that was the basis for the design used in the TI 8847.

TAYLOR, G. S. [1981]. "Compatible hardware for division and square root," *Proc. Fifth IEEE Symposium on Computer Arithmetic,* 127–134.

Good discussion of a radix-4 SRT division algorithm.

TAYLOR, G. S. [1985]. "Radix 16 SRT dividers with overlapped quotient selection stages," *Proc. Seventh IEEE Symposium on Computer Arithmetic,* 64–71.

Describes a very sophisticated high-radix division algorithm.

WESTE, N. AND K. ESHRAGHIAN [1985]. *Principles of CMOS VLSI Design,* Addison-Wesley, Reading, Mass.

This textbook has a section on the layouts of various kinds of adders.

WILLIAMS, T. E., M. HOROWITZ, R. L. ALVERSON, AND T. S. YANG [1987]. "A self-timed chip for division," *Advanced Research in VLSI, Proc. 1987 Stanford Conf.,* The MIT Press, Cambridge, Mass.

Describes a divider that tries to get the speed of a combinational design without using the area that would be required by one.

EXERCISES

A.1 [15/15/20] <A.3> Represent the following numbers as single-precision and double-precision IEEE floating-point numbers.

a. [15] 10

b. [15] 10.5

c. [20] 0.1

A.2 [10/15/20] <A.8> Complete the details of the block diagrams for the following adders.

a. [10] In Figure A.11, show how to implement the "1" and "2" boxes in terms of AND and OR gates.

b. [15] In Figure A.14, what signals need to flow from the adder cells in the top row into the "C" cells? Write the logic equations for the "C" box.

c. [20] Show how to extend the block diagram in A.13 so it will produce the carry-out bit c_8.

A.3 [15/15] <A.4> Floating-point addition.

a. [15] In a decimal system with $p = 5$, compute $-4.5673 + 4.9999 \times 10^{-5}$ assuming round to nearest. Give the value of the guard and round digits, and the sticky bit.

b. [15] What is the value of the sum for the other three rounding modes?

A.4 [15] <A.3> Show that if gradual underflow is not used, then it is no longer true that $x \neq y$ if and only if $x - y \neq 0$.

A.5 [25] <A.9> Write out the analogue of Figure A.21 for radix-8 Booth recoding.

A.6 [15] <A.3> Is the ordering of nonnegative floating-point numbers the same as integers when denormalized numbers are also considered? What if the denormalized numbers are represented using the wrapped representation mentioned in Section A.5?

A.7 [25/10] <A.2> One's complement.

a. [25] When adding two's complement numbers, you discard the carry out from the most significant bit. Show that in one's complement, you must add the carry back into the low end.

b. [10] Find the rule for detecting overflow in one's complement.

A.8 [15] <A.2> Equations A.2.1 and A.2.2 are for adding two n-bit numbers. Derive similar equations for subtraction, where there will be a borrow instead of a carry.

A.9 [15/20] <A.2> More one's complement.

a. [15] A complication that arises with one's complement arithmetic is that zero has two representations. Show that even if the negative form of zero is never an input, the adder in Equation A.2.1 (with c_0 the end around carry) can still produce a negative zero.

b. [20] Use the fact that $a + b = a - (-b)$ together with the subtractor circuit of the previous problem to derive a different one's complement adder. Can this adder ever produce negative zero?

A.10 [20] <A.2> On a machine that doesn't detect integer overflow in hardware, show how you would detect overflow on a signed addition operation in software.

A.11 [25] <A.9> In the array of Figure A.23, the fact that an array can be pipelined is not exploited. Can you come up with a design that feeds the output of the bottom CSA into the bottom CSAs instead of the top one, and that will run faster than the arrangement of Figure A.23?

A.12 [15] <A.9> For ordinary Booth recoding, the multiple of b used in the ith step is simply $a_{i-1} - a_i$. Can you find a similar formula for radix-4 Booth recoding (overlapped triplets)?

A.13 [25/15/30] <A.9> Shifting-over-zeros multiplication.

a. [25] Does Booth recoding always increase the number of zeros in a number? Can it ever decrease the number of zeros?

b. [15] Given the number $a_{n-1} \cdots a_0$, define $c_0 = 0$, and define c_i to be the carry out from adding a_i, a_{i-1}, and c_{i-1}. Then *modified Booth recoding* gives a number with digits $A_i = a_i + c_i - 2c_{i+1}$. What is the recoding of 01101?

c. [30] Show that modified Booth recoding never decreases the number of zeros.

A.14 [20/15/20/15/20/15] <A.6> Iterative square root.

a. [20] Use Newton's method to derive an iterative algorithm for square root. The formula will involve a division.

b. [15] What is the fastest way you can think of to divide a floating-point number by 2?

c. [20] If division is slow, then the iterative square root routine will also be slow. Use Newton's method on $f(x) = 1/x^2 - a$ to derive a method that doesn't use any divisions.

d. [15] Assume that the ratio division by 2 : floating-point add : floating-point multiply is 1:2:4. What ratios of multiplication time to divide time makes each iteration step in the method of Part c faster than each iteration in the method of Part a?

e. [20] When using the method of Part a, how many bits need to be in the initial guess in order to get double-precision accuracy after 3 iterations? (You may ignore rounding error.)

f. [15] Suppose that when Spice runs on the TI 8847, it spends 16.7% of its time in the square root routine (this percentage has been measured on other machines). Using the values in Figure A.31 and assuming 3 iterations, how much slower would Spice run if square root was implemented in software using the method of Part a?

A.15 [30/10] <A.2> This problem presents an algorithm for adding signed-magnitude numbers. If A and B are integers of opposite signs, let a and b be their magnitudes.

a. [30] Show that the following rules for manipulating the unsigned numbers a and b gives $A + B$

1. Complement one of the operands.
2. Using end around carry (as in the one's complement adder of problem A.7) add the complemented operand and the other (uncomplemented) one.
3. If there was a carry out, the sign of the result is the sign associated with the uncomplemented operand.
4. Otherwise, if there was no carry out, complement the result, and give it the sign of the complemented operand.

b. [10] <A.4> In our discussion of floating-point add, we suggested that when the result is negative the +1 needed to do two's complement be done in the rounding unit. Use the result of Part A to devise a floating-point adder that doesn't require this.

A.16 [15] <A.7> Our example that showed that double rounding can give a different answer from rounding once used the round-to-even rule. If halfway cases are rounded up, is double rounding still dangerous?

A.17 [15/30] <A.9> The text discussed radix-4 SRT division with quotient digits of –2, –1, 0, 1, 2. Suppose that 3 and –3 are also allowed as quotient digits.

a. [15] What relation replaces $|r_i| \leq 2b/3$?

b. [30] How many bits of b and P do you need to examine ?

A.18 [25] <A.6,A.9> The discussion of the remainder-step instruction assumed that division was done using a bit-at-a-time algorithm. What would have to change if division was implemented using a higher-radix method?

A.19 [20/20/25/25/20] <A.3> Signed-logarithm representation.

a. [20] Suppose you want to represent a number x by its sign and $\log|x|$. Then if $\log|x|$ is to be nonnegative, x must be ≥ 1. You can allow smaller x if you represent x by $\log k|x|$ for some constant k. Use 0 if $k|x| < 1$. Now $\log k|x|$ will not be an integer, but it can be represented as a fixed-point number. If we put the binary point m bits to the left of the least significant bit, write down formulas for converting x to signed-logarithm form and back.

b. [20] Give the rules for multiplication and division.

c. [25] Show that no matter what base of logs is used, this system cannot exactly represent all of 1, 2, and 3.

d. [25] Show how to implement addition using a table containing 2^{p-1} entries of $p - 1$ bits each, where the signed logarithm number is stored in a p-bit register.

e. [20] Show that for numbers which are exactly representable in this system, multiplication is exact, addition is not, but $a(b + c) = ab + ac$ exactly (when there is no over/underflow).

A.20 [20/10] <A.8> Carry-skip adders.

a. [20] Assuming that time is proportional to logic levels, what (fixed) block size gives the fastest addition for an adder of some fixed total length?

b. [10] Explain why the carry-skip adder takes time $\sqrt{n}$.

A.21 [Discussion] In the MIPS approach to exception handling, you need a test for determining whether two floating-point operands could cause an exception. This should be fast and also not have too many false positives. Can you come up with a practical test? The performance cost of your design will depend on the distribution of floating-point numbers. This is discussed in Knuth [1981] and Swartzlander [1980].

A.22 [35] <A.8> The simplest carry-select adder replaces an n-bit adder with $n/2$ bit adders and a mux. A more complex carry-select adder would use $n/4$-bit adders and more muxes. Can you design an adder that uses muxes and 1-bit adders and runs in O(log n) time? Such an adder is called a *conditional-sum adder.*

A.23 [10/15/20/15/15] <A.6> Correctly rounded iterative division. Let a and b be floating-point numbers with p-bit significands (p = 53 in double precision). Let q be the exact quotient $q = a/b$. Suppose that $\bar{q}$ is the result of an iteration process, that $\bar{q}$ has a few extra bits of precision, and that $0 < q - \bar{q} < 2^{-p}$.

a. [10] If x is a floating-point number, and $1 \le x < 2$, what is the next representable number after x?

b. [15] Show how to compute q' from $\bar{q}$, where q' has $p + 1$ bits of precision and $|q - q'| < 2^{-p}$.

c. [20] Assuming round to nearest, show that the correctly rounded quotient is either q', $q' - 2^{-p}$, or $q' + 2^{-p}$.

d. [15] Give rules for computing the correctly rounded quotient from q' based on the low- order bit of q' and the sign of $a - bq'$.

e. [15] Solve Part c for the other three rounding modes.

B Complete Instruction Set Tables

B.1 VAX User Instruction Set

The following tables include all the VAX user instructions; the system instructions are not included.

The underscore following the instruction name implies that the instruction will operate upon any data type contained in the parentheses following that instruction. The data type abbreviations are:

B = byte (8 bits)	F = F_floating (32 bits)
W = word (16 bits)	D = D_floating (64 bits)
L = longword (32 bits)	G = G_floating (64 bits)
Q = quadword (64 bits)	H = H_floating (128 bits)
O = octaword (128 bits)	

Integer and Floating-Point Logical and Arithmetic Instructions

Instruction	Description
ADAWI	Add aligned word interlocked
ADD_2	Add (B,W,L,F,D,G,H) 2 operand
ADD_3	Add (B,W,L,F,D,G,H) 3 operand
ADWC	Add with carry
ASH_	Arithmetic shift (L,Q)
BIC_2	Bit clear (B,W,L) 2 operand
BIC_3	Bit clear (B,W,L) 3 operand
BICPSW	Bit clear processor status word
BIS_2	Bit set (B,W,L) 2 operand
BIS_3	Bit set (B,W,L) 3 operand
BISPSW	Bit set processor status word
BIT_	Bit test (B,W,L)
CLR_	Clear (B,W,L=F,Q=D=G,O=H)
CVT_	Convert (B,W,L,F,D,G,H)(B,W,L,F,D,G,H) except BB, WW, LL, FF, DD, GG, HH, DG, and GD
CVTR_L	Convert rounded (F,D,G,H) to longword
CMP_	Compare (B,W,L,F,D,G,H)
DEC_	Decrement (B,W,L)
DIV_2	Divide (B,W,L,F,D,G,H) 2 operand
DIV_3	Divide (B,W,L,F,D,G,H) 3 operand
EDIV	Extended divide
EMOD_	Extended modulus (F,D,G,H)
EMUL	Extended multiply

Instruction	Description
INC_	Increment (B,W,L)
INDEX	Compute index
MCOM_	Move complemented (B,W,L)
MNEG_	Move negated (B,W,L,F,D,G,H)
MOVA_	Move address (B,W,L=F,Q=D=G,O=H)
MOV_*	Move (B,W,L,F,D,G,H,Q,O)**—general move between two operands
MOVPSL	Move from processor status longword
MOVZ_	Move zero-extended (BW,BL,WL)
MUL_2	Multiply (B,W,L,F,D,G,H) 2 operand
MUL_3	Multiply (B,W,L,F,D,G,H) 3 operand
POLY_	Polynomial evaluation (F,D,G,H)
POPR	Pop registers from stack
PUSHA_	Push address (B,W,L=F,Q=D=G,O=H) on stack
PUSHL	Push longword on stack
PUSHR	Push registers on stack
ROTL	Rotate longword
SBWC	Subtract with carry
SUB_2	Subtract (B,W,L,F,D,G,H) 2 operand
SUB_3	Subtract (B,W,L,F,D,G,H) 3 operand
TST_	Test (B,W,L,F,D,G,H)
XOR_2	Exclusive or (B,W,L) 2 operand
XOR_3	Exclusive or (B,W,L) 3 operand

Branch, Jump, and Procedure Call Instructions

Instruction	Description
ACB_	Add, compare and branch (B,W,L.F,D,G,H)
AOBLEQ	Add one and branch less than or equal
AOBLSS	Add one and branch less than
BB_	Branch on bit (set, clear)
BBS_	Branch on bit (set, clear) and (set, clear) bit
BB_I	Branch on bit set (clear) and set (clear) bit interlocked
BCC	Branch carry cleared
BCS	Branch carry set
BEQL	Branch equal
BEQLU	Branch equal unsigned
BGEQ	Branch greater than or equal
BGEQU	Branch greater than or equal unsigned
BGTR	Branch greater than

Instruction	Description
BGTRU	Branch greater than unsigned
BLB_	Branch on low bit (set, clear)
BLEQ	Branch less than or equal
BLEQU	Branch less than or equal unsigned
BLSS	Btranch less than
BLSSU	Branch less than unsigned
BNEQ	Branch not equal
BNEQU	Branch not equal unsigned
BR_	Jump with (B,W) displacement
BSB_	Branch to subroutine with (B,W) displacement
BV_	Branch overflow (set,clear)
CALLG	Call procedure with general argument list
CALLS	Call procedure with stack argument list
CASE_	Case on (B,W,L)
JMP	Jump
JSB	Jump to subroutine
RET	Return from procedure
RSB	Return from subroutine
SOBGEQ	Subtract one and branch greater than or equal
SOBGTR	Subtract one and branch greater than

Decimal and String Instructions

Instruction	Description
ADDP4	Add packed 4 operand
ADDP6	Add packed 6 operand
ASHP	Arithmetic shift packed and round
CMPC3	Compare characters 3 operand
CMPC5	Compare characters 5 operand
CMPP3	Compare packed 3 operand
CMPP4	Compare packed 4 operand
CRC	Calculate cyclic redundancy check
CVTLP	Convert long to packed
CVTPL	Convert packed to long
CVTPT	Convert packed to trailing
CVTTP	Convert trailing to packed
CVTPS	Convert packed to separate
CVTSP	Convert separate to packed
DIVP	Divide packed
EDITPC	Edit packed to character string

Instruction	Description
LOCC	Locate character
MATCHC	Match characters
MOVC3	Move character 3 operand
MOVC5	Move character 5 operand
MOVP	Move packed
MOVTC	Move translated characters
MOVTUC	Move translated until character
MULP	Multiply packed
SCANC	Scan characters
SKPC	Skip character
SPANC	Span characters
SUBP4	Subtract packed 4 operand
SUBP6	Subtract packed 6 operand

Variable-Length Bit Field Instructions

Instruction	Description
CMPV	Compare field
CMPZV	Compare zero-extended field
EXTV	Extract field
EXTZV	Extract zero-extended field
INSV	Insert field
FFS	Find first set
FFC	Find first clear

Queue Instructions

Instruction	Description
INSQHI	Insert entry into queue at head, interlocked
INSQTI	Insert entry into queue at tail, interlocked
INSQUE	Insert entry in queue
REMQHI	Remove entry from queue at head, interlocked
REMQTI	Remove entry from queue at tail, interlocked
REMQUE	Remove entry from queue

B.2 System/360 Instruction Set

The 360 instruction set is shown in the following tables, organized by instruction type and format. System/370 contains 15 additional user instructions.

Integer/Logical and Floating-Point R–R Instructions

The * indicates the instruction is floating point, and may be either D (double precision) or E (single precision).

Instruction	Description
ALR	Add logical register
AR	Add register
A*R	FP addition
CLR	Compare logical register
CR	Compare register
C*R	FP compare
DR	Divide register
D*R	FP divide
H*R	FP halve
LCR	Load complement register
LC*R	Load complement
LNR	Load negative register
LN*R	Load negative
LPR	Load positive register
LP*R	Load positive
LR	Load register
L*R	Load FP register
LTR	Load and test register
LT*R	Load and test FP register
MR	Multiply register
M*R	FP multiply
NR	And register
OR	Or register
SLR	Subtract logical register
SR	Subtract register
S*R	FP subtraction
XR	Exclusive or register

Branches and Status Setting R–R Instructions

These are R–R format instructions that either branch or set some system status; several of them are privileged and legal only in supervisor mode.

Instruction	Description
BALR	Branch and link
BCTR	Branch on count
BCR	Branch/condition
ISK	Insert key
SPM	Set program mask
SSK	Set storage key
SVC	Supervisor call

Integer/Logical and Floating-Point Instructions—RX Format

These are all RX format instructions. The symbol "+" means either a word operation (and then stands for nothing) or H (meaning halfword); for example, A+ stands for the two opcodes A and AH. The symbol "*" is D or E standing for double- or single-precision floating point.

Instruction	Description
A+	Add
A*	FP add
AL	Add logical
C+	Compare
C*	FP compare
CL	Compare logical
D	Divide
D*	FP divide
L+	Load
L*	Load FP register
M+	Multiply
M*	FP multiply
N	And
O	Or
S+	Subtract
S*	FP subtract
SL	Subtract logical
ST+	Store
ST*	Store FP register
X	Exclusive or

Branches and Special Loads and Stores—RX format

Instruction	Description
BAL	Branch and link
BC	Branch condition
BCT	Branch on count
CVB	Convert-binary
CVD	Convert-decimal
EX	Execute
IC	Insert character
LA	Load address
STC	Store character

RS and SI Format Instructions

These are the RS and SI format instructions. The symbol "*" may be A (arithmetic) or L (logical).

Instruction	Description
BXH	Branch/high
BXLE	Branch/low-equal
CLI	Compare logical immediate
HIO	Halt I/O
LPSW	Load PSW
LM	Load multiple
MVI	Move immediate
NI	And immediate
OI	Or immediate
RDD	Read direct
SIO	Start I/O
SL*	Shift left A/L
SLD*	Shift left double A/L
SR*	Shift right A/L
SRD*	Shift right double A/L
SSM	Set system mask
STM	Store multiple
TCH	Test channel
TIO	Test I/O
TM	Test under mask
TS	Test and set
WRD	Write direct
XI	Exclusive or immediate

SS Format Instructions

These are all decimal or string instructions.

Instruction	Description
AP	Add packed
CLC	Compare logical chars
CP	Compare packed
DP	Divide packed
ED	Edit
EDMK	Edit and mark
MP	Multiply packed
MVC	Move character
MVN	Move numeric
MVO	Move with offset
MVZ	Move zone
NC	And characters
OC	Or characters
PACK	Pack (Character → decimal)
SP	Subtract packed
TR	Translate
TRT	Translate and test
UNPK	Unpack
XC	Exclusive or characters
ZAP	Zero and add packed

B.3 8086 Instruction Set

These charts contain the instruction set of the 8086; floating-point instructions that are neither included nor used by the 8086 benchmarks are not included.

Arithmetic and Logical Instructions

Instruction	Description
AAA	ASCII adjust after addition
AAD	ASCII adjust before division
AAM	ASCII adjust after multiplication
AAS	ASCII adjust after subtraction
ADC	Add with carry
ADD	Integer addition
AND	Logical and
CBW/CWD/CDQ	Convert byte to word/word to dword/dword to quad
CLC	Clear the carry flag
CLD	Clear the direction flag
CLI	Clear the interrupt flag
CMC	Complement the carry flag
CMP	Compare
DAA	Decimal adjust after addition
DAS	Decimal adjust after subtraction
DEC	Decrement
DIV	Unsigned divide
IDIV	Signed divide
IMUL	Signed multiplication
INC	Increment
MUL	Unsigned multiplication
NEG	Negate
NOT	Not
OR	Inclusive or
RCL	Rotate through carry left
RCR	Rotate through carry right
ROL	Rotate left
ROR	Rotate right
SAL/SHL	Shift arithmetic left
SAR	Shift arithmetic right
SBB	Subtract with borrow
SHR	Shift logical right
STC	Set carry flag
STD	Set direction flag
STI	Set interrupt flag
SUB	Subtract
TEST	Logical compare
XOR	Exclusive or

Control Instructions

Instruction	Description
CALL	Call procedure (intrasegment)
CALL	Call procedure (intersegment)
HLT	Halt
INT	Call to interrupt procedure
INTO	On overflow call interrupt procedure
IRET	Interrupt return
JB/JNAE/JC	Jump below
JBE/JNA	Jump below or equal
JCXZ/JECXZ	Jump CX/ECX zero
JE/JZ	Jump equal
JL/JNGE	Jump less
JLE/JNG	Jump less or equal
JMP	Jump (intrasegment)
JMPF	Jump (intersegment)
JNB/JAE/JNC	Jump not below
JNBE/JA	Jump not below or equal
JNE/JNZ	Jump not equal
JNL/JCE	Jump not less
JNLE/JG	Jump not less or equal
JNO	Jump no overflow
JNP/JPO	Jump not parity
JNS	Jump not sign
JO	Jump overflow
JP/JPE	Jump parity
JS	Jump sign
LOCK	Bus lock
RET	Return (intrasegment)
RETF	Return (intersegment)

Data Transfer Instructions

Instruction	Description
IN	Input from a port
LAHF	Load flags into AH register
LDS	Load pointer to DS
LEA	Load effective address
LES	Load pointer to ES
LOCK	Bus lock
MOV	Move
OUT	Output to a port
POP	Pop off stack
POPF/POPFD	Pop from stack into flags
PUSH	Push onto stack
PUSH	Push segment register onto the stack
PUSHF/PUSHFD	Push flags onto stack
SAHF	Store AH register into flags
XCHC	Exchange
XLAT/XLATB	Table lookup translation

String Instructions

Instruction	Description
CMPS/CMPSB/CMPSW/CMPSD	Compare string
LODS/LODSB/LODSW/LODSD	Load string
MOVS/MOVSB/MOVSW/MOVSD	Move string
REP	Repeat
REPE/REPZ	Repeat while equal
REPNE/REPNZ	Repeat while not equal
SCAS/SCASB/SCASW/SCASD	Scan string
STOS/STOSB/STOSW/STOSD	Store string

C Detailed Instruction Set Measurements

C.1 VAX Detailed Measurements

Instruction	GCC	Spice	TeX	COBOLX	Average
Control	**30%**	**18%**	**30%**	**25%**	**26%**
Conditional Branch	20%	13%	19%	18%	17%
BRB,BRW	6%	3%	4%	5%	5%
CALLS,CALLG	2%	1%	4%	0%	2%
RET	2%	1%	4%	0%	2%
JMP				2%	1%
Arithmetic, logical	**40%**	**23%**	**33%**	**24%**	**30%**
CMP*	12%	5%	11%	9%	9%
ADDL_	5%	12%	4%		5%
INCL	3%		3%	5%	3%
MOVA*	1%	3%	4%	2%	3%
TSTL	4%	2%	3%		2%
CLRL	3%	1%	2%	3%	2%
SUB*_	3%	1%	3%		2%
CVT*L	6%			0%	2%
ASHL	3%		3%	0%	2%
MULL_	0%			5%	1%
Data transfer	**19%**	**15%**	**28%**	**4%**	**16%**
MOVL	15%	9%	17%	4%	11%
PUSHL	3%		7%		2%
MOVQ		6%			1%
MOVZ*L	1%		4%		1%
Floating point	**0%**	**23%**	**0%**	**0%**	**6%**
MULD_		9%			2%
SUBD_		6%			1%
ADDD_		6%			1%
DIVD_		3%			1%
CMPD		2%			
Decimal, string	**0%**	**0%**	**1%**	**38%**	**10%**
CVTTP,CVTPT				19%	5%
MOVC3,MOVC5			1%	9%	2%
ADDP4				6%	1%
CMPP_				2%	1%
CMPC3				2%	1%
Totals	**88%**	**79%**	**92%**	**88%**	**87%**

FIGURE C.1 Instructions responsible for more than 1.5% of the dynamic executions in any benchmark. The instructions are broken into five classes, printed in boldface. The data in those rows give the total frequency for the operations in that class. Cells representing a contribution of 1% or less are empty, except the average column can have an entry of 1%. Because of rounding, the average can differ from what might appear to be correct if based on the figures in the individual columns.

C.2 360 Detailed Measurements

Instruction	PLIC	FORTGO	PLIGO	COBOLGO	Average
Control	**32%**	**13%**	**5%**	**16%**	**16%**
BC,BCR	28%	13%	5%	14%	15%
BAL,BALR	3%			2%	1%
Arithmetic, logical	**29%**	**35%**	**29%**	**9%**	**26%**
A,AR	3%	17%	21%		10%
SR	3%	7%			3%
SLL		6%	3%		2%
LA	8%	1%	1%		2%
CLI	7%				2%
NI				7%	2%
C	5%	4%	4%	0%	3%
TM	3%	1%		3%	2%
MH			2%		1%
Data transfer	**17%**	**40%**	**56%**	**20%**	**33%**
L,LR	7%	23%	28%	19%	19%
MVI	2%		16%	1%	5%
ST	3%		7%		3%
LD		7%	2%		2%
STD		7%	2%		2%
LPDR		3%			1%
LH	3%				1%
IC	2%				1%
LTR		1%			0%
Floating point		**7%**			**2%**
AD		3%			1%
MDR		3%			1%
Decimal, string	**4%**			**40%**	**11%**
MVC	4%			7%	3%
AP				11%	3%
ZAP				9%	2%
CVD				5%	1%
MP				3%	1%
CLC				3%	1%
CP				2%	1%
ED				1%	0%
Total	**82%**	**95%**	**90%**	**85%**	**88%**

FIGURE C.2 (See previous page.) Distribution of instruction execution frequencies for the four 360 programs. All instructions with a frequency of execution greater than 1.5% are included. Immediate instructions, which operate on only a single byte, are included in the section that characterizes their operation, rather than with the long character-string versions of the same operation. By comparison, the average frequencies for the major instruction classes of the VAX are 23% (control), 28% (arithmetic), 29% (data transfer), 7% (floating point), and 9% (decimal). Once again, a 1% entry in the average column can occur because of entries in the constituent columns.

C.3 Intel 8086 Detailed Measurements

Instruction	Turbo C	MASM	Lotus	Average
Control	**21%**	**20%**	**32%**	**24%**
Conditional jumps	10%	12%	9%	10%
CALL,CALLF	4%	3%	5%	4%
RET,RETF	4%	3%	5%	4%
LOOP			12%	4%
JMP	3%	2%	2%	2%
Arithmetic, logical	**23%**	**24%**	**26%**	**25%**
CMP	8%	9%	5%	7%
SAL,SHR,RCR	2%	1%	11%	5%
ADD	3%	2%	3%	3%
OR, XOR	4%	2%	2%	3%
INC, DEC	3%	4%	3%	3%
SUB	2%	3%		2%
CBW	1%	1%		1%
TEST		2%	2%	1%
Data transfer	**49%**	**46%**	**30%**	**42%**
MOV	29%	31%	21%	27%
LES	6%	2%		3%
PUSH	10%	8%	4%	7%
POP	5%	6%	5%	5%
Totals	**93%**	**90%**	**88%**	**90%**

FIGURE C.3 The instructions responsible for more than 1.5% of the executions on any of the three benchmarks Some very similar instructions were combined for simplicity. Although MASM makes some use of string operations, the frequency is too low to make the table.

C.4 DLX Detailed Instruction Set Measurements

Instruction	GCC	Spice	TeX	US Steel	Average
Control	**21%**	**5%**	**7%**	**23%**	**14%**
B--Z	19%	2%	7%	16%	11%
J	2%	3%		3%	2%
JAL				2%	0%
JR				2%	0%
Arithmetic, logical	**37%**	**28%**	**41%**	**49%**	**39%**
ADDU,ADDUI	17%	16%	20%	27%	20%
LHI	2%	7%	10%	3%	5%
SLL	5%	5%	5%	4%	5%
LI	4%		4%	6%	4%
S--,S--I	5%		3%	3%	3%
AND,ANDI	2%			3%	1%
SRA	2%			2%	1%
OR,ORI				2%	1%
Data transfer	**28%**	**35%**	**33%**	**10%**	**26%**
LW	18%	8%	19%	5%	13%
SW	10%	2%	12%	5%	7%
LBU			2%		1%
LD		14%			4%
SD		6%			1%
MOVFP2I, MOVI2FP		5%			1%
Floating point	**0%**	**15%**	**0%**	**0%**	**4%**
FMUL		5%			1%
FADD		4%			1%
FSUB		3%			1%
FDIV		3%			1%
Totals	**85%**	**83%**	**82%**	**82%**	**83%**

FIGURE C.4 Instruction mixes for GCC, Spice, TeX, and the U.S. Steel COBOL benchmark. Some instructions were combined, both in the interest of space and because the combined class more correctly reflects what the processor is doing. The instruction class "B--Z" includes all conditional branches (which are all compares to zero). The class "S--,S--I" includes all set conditional instructions, both immediate and register–register. Immediate operations have been combined with the non-immediate class for all operations except loads, where they are distinctly different. Again, a blank space means that the instruction is not responsible for more than 1.5% of the executions, and the average may appear at 1% or less because the instruction is not used by all benchmarks.

D Time Versus Frequency Measurements

D.1 Time Distribution on the VAX-11/780

We know from Chapters 2 and 3 that measuring instruction counts alone can be misleading. In this appendix we will examine the time distributions for some programs running on these four machines. For the 360, the 8086, and DLX, we will show the time distribution averaged over the three programs in the graph format used earlier. For the VAX, we will use measurements reported in Clark and Levy [1982] (see References in Chapter 4).

Figure D.1 shows the distribution of instruction executions, both by time and by frequency of occurrence. These data were measured by Emer and reported by Clark and Levy for a VAX-11/780 running VMS with multiple users doing three primary tasks:

1. Updating indexed files
2. Executing a matrix multiplication routine
3. Doing program development, including editing, compiling, and debugging

Figure D.1 includes any user instruction that accounts for more than 1% of the instruction executions or more than 1% of execution time. There are 26 instructions that fit this description, and together they account for 59% of the executions and 58% of the time. The measured data include the operating system and file system overhead.

Time distributions are particularly important on architectures like the VAX, where the number of cycles for an instruction may vary from one or two up to tens or hundreds.

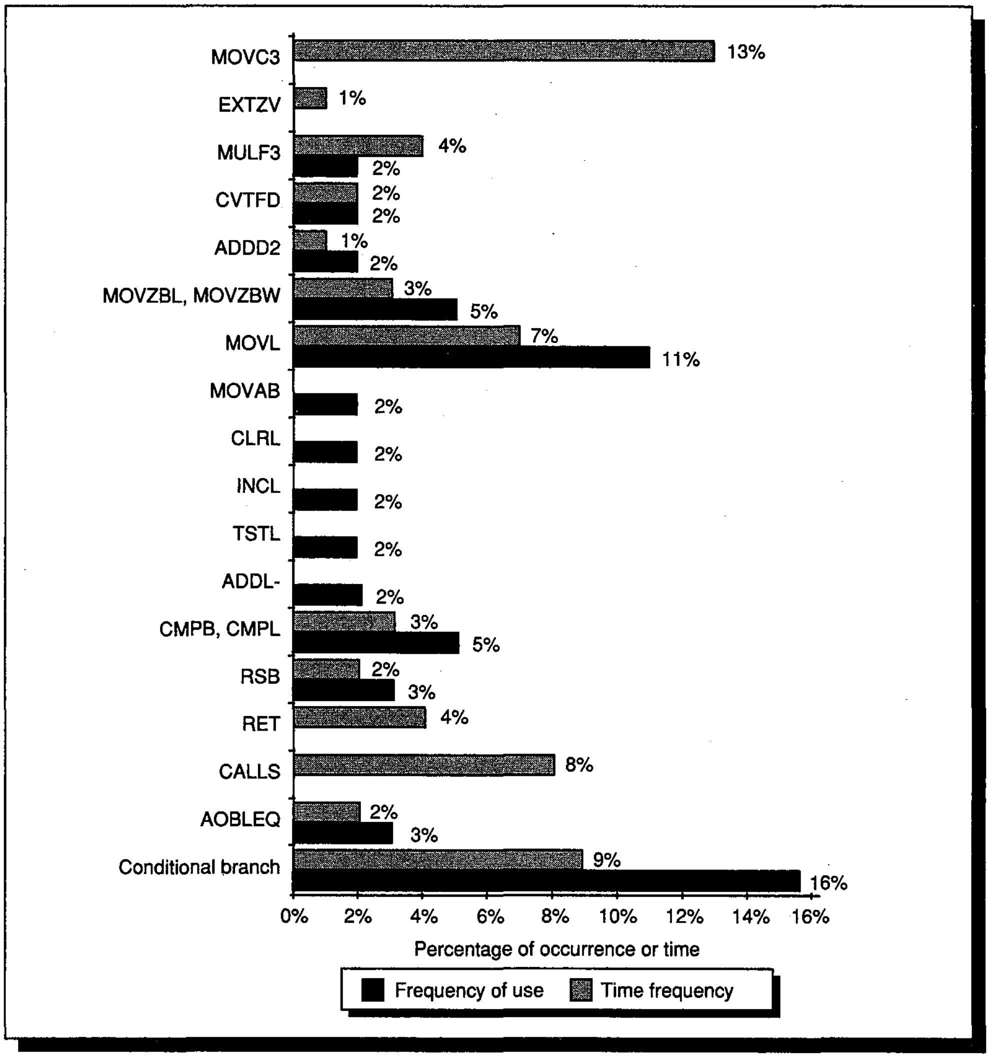


FIGURE D.1 Time and frequency distribution for a multiuser workload on a VAX-11/780 running VMS. This data includes all user instructions that are responsible for more than 1% of either the instruction executions or the execution time. (Two operating system instructions (REI and MTPR), each of which accounts for about 1% of the execution time, are not included.) The absence of an execution-frequency bar or time-frequency bar for an entry (such as MOVC3 or TSTL) means that the time frequency or execution-time frequency is below 1% (not that it is 0!). Clark and Levy [1982] commented that the large percentage of time consumed by the MOVC3 in the time distribution is somewhat abnormal for a nonbusiness workload and has not been observed in other measurements on the 11/780.

D.2 Time Distribution on the IBM 370/168

Figure D.2 shows the time distribution on an IBM 370/168 for the same programs we discussed in Chapter 4 and included in Figure 4.28 (page 175). All instructions that are responsible for more than 1.5% of the execution frequency and the execution time for at least one program are included. Several

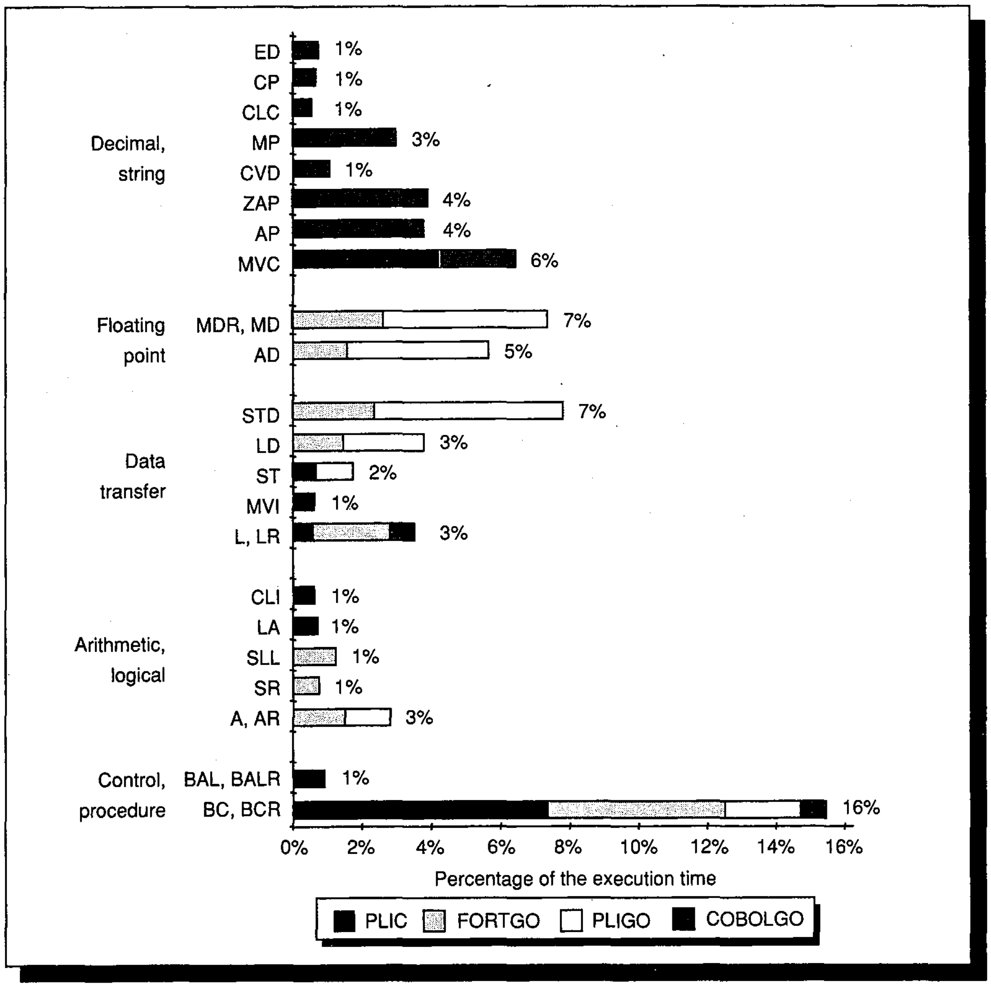


FIGURE D.2 Time distribution for the four programs discussed in Chapter 4 running on an IBM 370/168. The corresponding data on execution frequency appears in Figure 4.28 (page 175), or in table form in Figure C.2. Any instruction with greater than 1.5% frequency in the time distribution and in the execution-count distribution is included in this chart. Shustek [1978] (see References in Chapter 4) computed these numbers using a model of the 370/168 CPU. The model predicts the execution time for the programs and has an overall accuracy for each program of about 99% except on PLIGO, where it has an 8% error.

instructions appeared in the time distribution that were not in the frequency distribution, where their occurrence was too low. These instructions, which are not in Figure 4.28, are

TRT—Translate and test, a string instruction used by the PL/I compiler, most likely to scan the input source; takes 5.4% of the time in that program.

DP—Divide packed, a low frequency but long-running instruction that takes 18.7% of the time in COBOLGO.

DDR—Divide double register, a floating-point divide, infrequent but long running at 5.2% of the FORTGO execution time.

LM and STM—Load multiple and store multiple, with frequencies just below 1%, are somewhat slower than the average instruction; thus, they take 3% to 4% of the cycles in PLIGO.

BCT,BXLE—Loop branches that involve incrementing counts or doing other compares; BCT consumes about 2% of the time in PLIC, and BXLE consumes 3.5% in FORTGO.

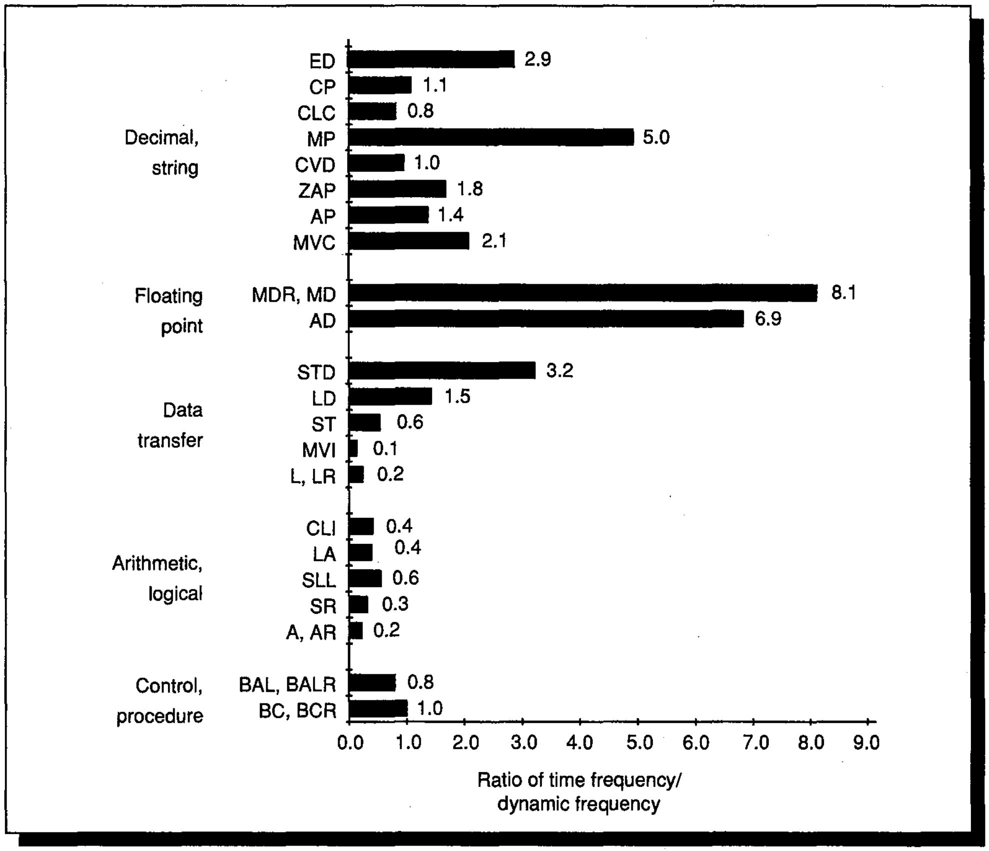


FIGURE D.3 Time frequency (percent of cycles doing this instruction as measured on an IBM 370/168) divided by dynamic frequency (percent of executions for this instruction). The programs are those in Chapter 4. This data is obtained directly from Figures 4.28 (page 175) and Figure D.2. This clearly shows that the floating-point instructions are the most expensive.

Several of the simpler but lower-frequency data transfer and ALU instructions that appeared in the frequency distribution do not appear in the time distribution because they constitute a very small percentage of the execution time. In total, the instructions shown in Figure D.2 account for 89% of the instruction executions and 72% of the execution time.

Figure D.3 gives the average execution time divided by the average frequency for those instructions that appear in both distributions. This measurement is a ratio that indicates the relative cost of an instruction. For example, an instruction that is responsible for 10% of the executions and 10% of the execution time will have a ratio of 1:1, or a cost factor of 1, and a CPI equal to the average CPI on the machine.

D.3 Time Distribution on an 8086 in an IBM PC

Figure D.4 continues our examination of time distribution by looking at the top time-consuming instructions on the 8086 for the same programs as measured in Chapter 4. These curves look very similar to those in Figure 4.32 (page 178), the frequency distribution for the 8086 (shown in table form in Figure C.3, page C-4). Two arithmetic and logical instructions, `CBW` and `SUB`, that appeared in the frequency distribution do not appear in the top of the execution-time distribution. Additionally, there are four instructions that have a significant contribution to the time frequency but are not in the execution-frequency distribution:

- String instructions `SCAS` (a string search) and `MOVS` (a string move). Both instructions are used in MASM, where they account for 8% and 7% of the execution time, respectively. `MOVS` is also used in Lotus, where it accounts for 6.6% of the program's execution time.
- Integer multiply and divide `ML16` and `DV16`. These are used in Lotus, where they respectively account for 10% and 4% of the program's execution time.

Together, the instructions in Figure D.4 are responsible for 87% of the instruction executions and 85% of the execution time.

Figure D.5 shows the ratio of execution time to execution frequency in the same fashion used for the IBM 360. Calls, returns, and loading a segment register consume a larger percentage of the execution time relative to their dynamic occurrence. However, the overall execution time profile of the 8086 is much closer to the execution frequency profile—the correspondence is often 1:1, and never as high as 1:2. This is primarily because the variation in CPI among instructions is small compared to an overall average CPI of 14.1. The long-running instructions that do not even appear in the frequency counts but are major consumers of execution time (and would have a high CPI) are the string instructions and integer multiply and divide.

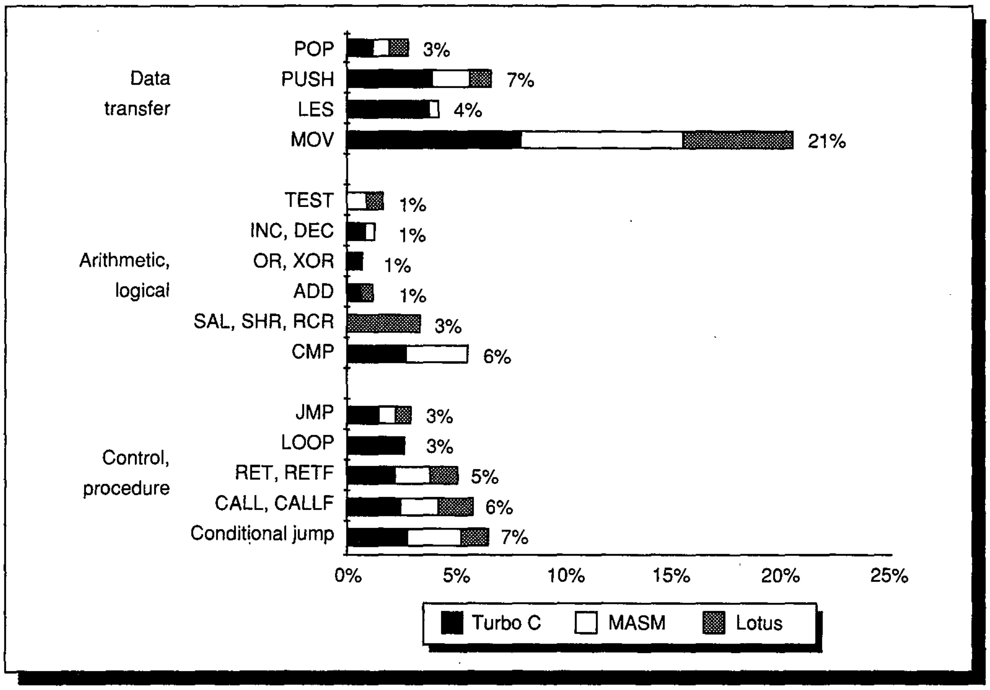


FIGURE D.4 The 8086 time distribution as measured on an IBM PC running MS-DOS. The format and data are the same as in Figure 4.32 (page 178).

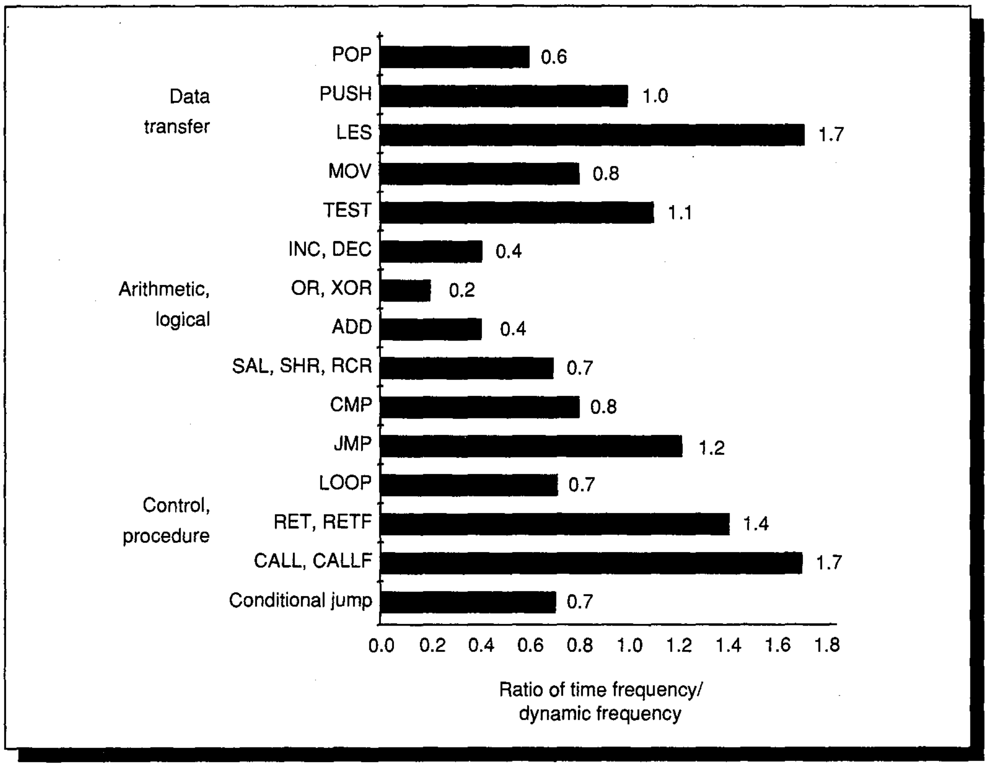


FIGURE D.5 Time distribution divided by frequency distribution for the 8086. This data is directly derived from Figures 4.32 (page 178) and D.4. The distribution is remarkably flatter than that for the IBM 360 or the VAX.

D.4 Time Distribution on a DLX Relative

To obtain a time distribution for DLX, we turn to the DECstation 3100, which has an instruction set architecture very similar to DLX (see Appendix E). The time distribution on the DECstation 3100 for the same programs measured in Chapter 4 (Figure 4.34 on page 181 and in table form in Figure C.4 is shown in Figure D.6. Figure D.6 includes all instructions that contribute more than 1% to the execution time. In total, these instructions account for 81% of all instruction executions and 97% of the execution time.

This time distribution is by far the closest to the frequency distribution. This is because under ideal conditions almost all instructions in DLX can take one cycle; only the LD and SD instructions must take two cycles. Of course, these perfect conditions never arise. The average CPI using the DECstation 3100 as a base is about 1.6 for GCC, TeX, and COBOLX, and about 2.1 for Spice.

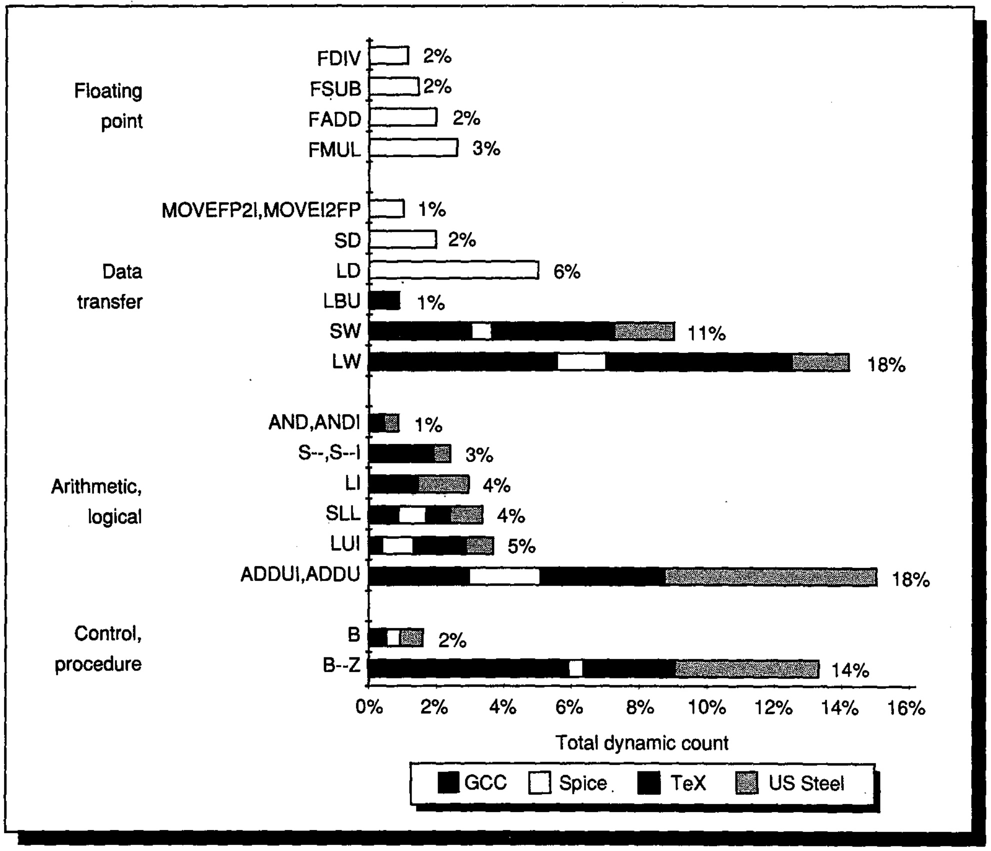


FIGURE D.6 The time distribution for our three benchmarks plus the US Steel COBOL benchmark as they would run on DLX using the CPI measurements from a DECstation 3100.

Figure D.7 shows contribution to execution time over contribution to execution frequency for the top instructions. Like the 360 and 8086 charts, a value above 1 indicates that this instruction has a higher CPI than the average instruction. Remember, though, that the ratio does not indicate the CPI for the instruction. However, we can use this figure to find the CPI for an instruction, given the base CPI for a specific program.

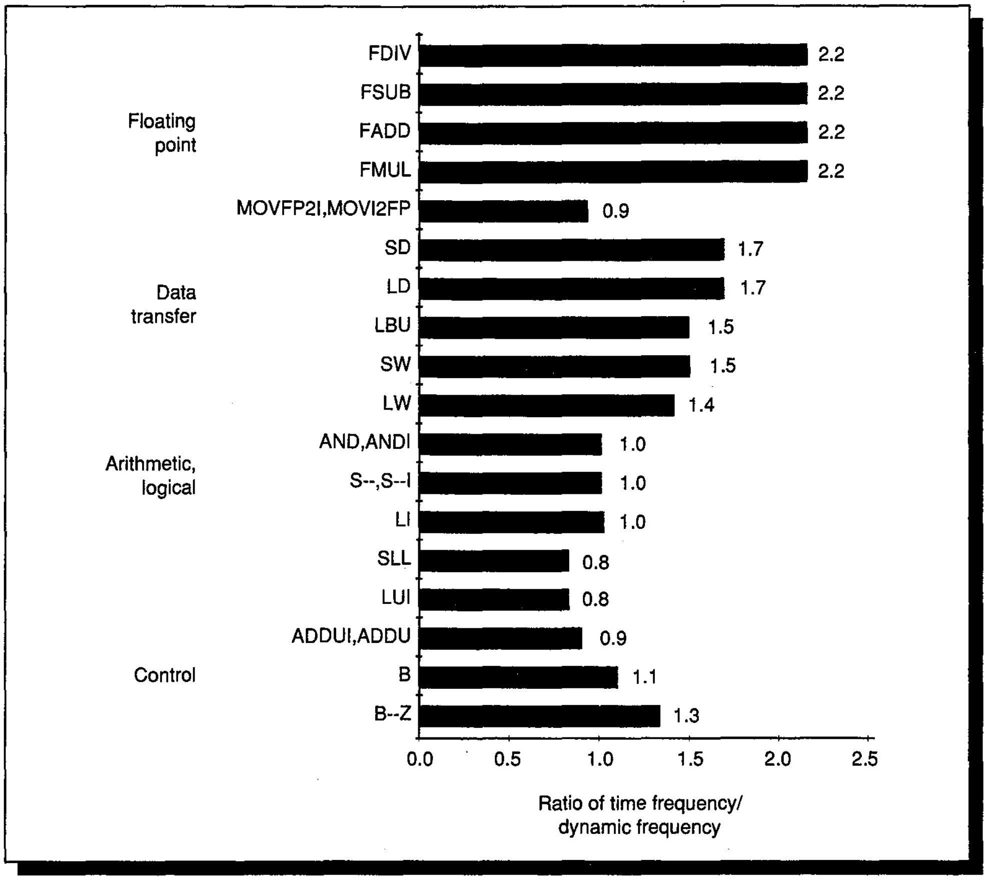


FIGURE D.7 Time frequency divided by execution frequency for DLX as measured using the time data from Figure D.6 and the frequency data from Figure 4.34 (page 181). The integer register–floating-point register moves are inexpensive, since they are really register–register operations. Surprisingly, the double-precision memory references are **not** twice as expensive as the 32-bit loads and stores. Can you hypothesize why based on the discussions of pipelining and cache design?

RISC: any computer announced after 1985.

Steven Przybylski (a designer of the Stanford MIPS)

E Survey of RISC Architectures

E.1 Introduction

We cover four examples of Reduced Instruction Set Computer (RISC) architectures in this appendix:

- Intel 860;
- MIPS R3000/R3010 (plus a section on MIPS II, used in the R6000);
- Motorola M88000; and
- SPARC, developed originally by Sun Microsystems.

We also include DLX, the instruction set architecture invented for this book. (A review of DLX can be found in the back inside cover or in pages 160–167 of Chapter 4.) Characteristics of these architectures are found in Figure E.1.

There has never been another class of computers that were so similar. This similarity allows the presentation of four architectures at once, with DLX thrown in for good measure! After presenting the addressing modes and instruction formats, the instructions are presented in three steps:

- Instructions found in DLX;
- Instructions not found in DLX but found in two or more architectures; and
- The unique instructions and characteristics of each architecture.

We conclude with a speculation about the future directions for RISCs.

	DLX	i860	MIPS	M88000	SPARC
Date announced	1990	1989	1986	1988	1987
Instruction size (bits)	32	32	32	32	32
Address space (size, model)	32 bits, flat	32 bits, flat	32 bits, flat	32 bits, flat	32 bits, flat
Data alignment	Aligned	Aligned	Aligned	Aligned	Aligned
Data addressing modes	1	2	1	3	2
Protection	Page	Page	Page	Page	Page
Page size	4 KB	4 KB	4 KB	4 KB	4–64 KB
I/O	Memory mapped	Memory mapped	Memory mapped	Memory mapped	Memory mapped
Integer registers (size, model, number)	31 GPR x 32 bits	31 GPR x 32 bits	31 GPR x 32 bits	31 GPR x 32 bits	31 GPR x 32 bits
Separate floating-point registers	32 x 32 or 16 x 64 bits	30 x 32 or 15 x 64 bits	16 x 32 or 16 x 64 bits	0	32 x 32 or 16 x 64
Floating-point format	IEEE 754 single, double	IEEE 754 single, double	IEEE 754 single, double	IEEE 754 single, double	IEEE 754 single,double

FIGURE E.1 Summary of five recent architectures. Except for number of data address modes and some instruction set details, the integer instruction sets of these architectures of the late 1980s are identical. Contrast this to Figure E.13, page E-23.

E.2 Addressing Modes and Instruction Formats

Figure E.2 shows the data addressing modes supported by each architecture. Since all have one register that always has the value 0—in fact, it is `r0` in every architecture—the absolute address mode with limited range can be synthesized using `r0` as the base in displacement addressing. Similarly, register-indirect addressing is synthesized by using displacement addressing with an offset of 0. Simplified addressing modes is one distinguishing feature between these and prior architectures.

Addressing mode	DLX	i860	MIPS	M88000	SPARC
Register + offset (displacement or based)	√	√	√	√	√
Register + register (indexed)	--	√	--	√	√
Register + scaled register (scaled)	--	--	--	√	--

FIGURE E.2 Summary of data addressing modes. (These addressing modes are explained in Section 3.4, pages 94–103) While the i860 does have indexed data addressing for all loads and floating-point stores, it is not available for integer stores.

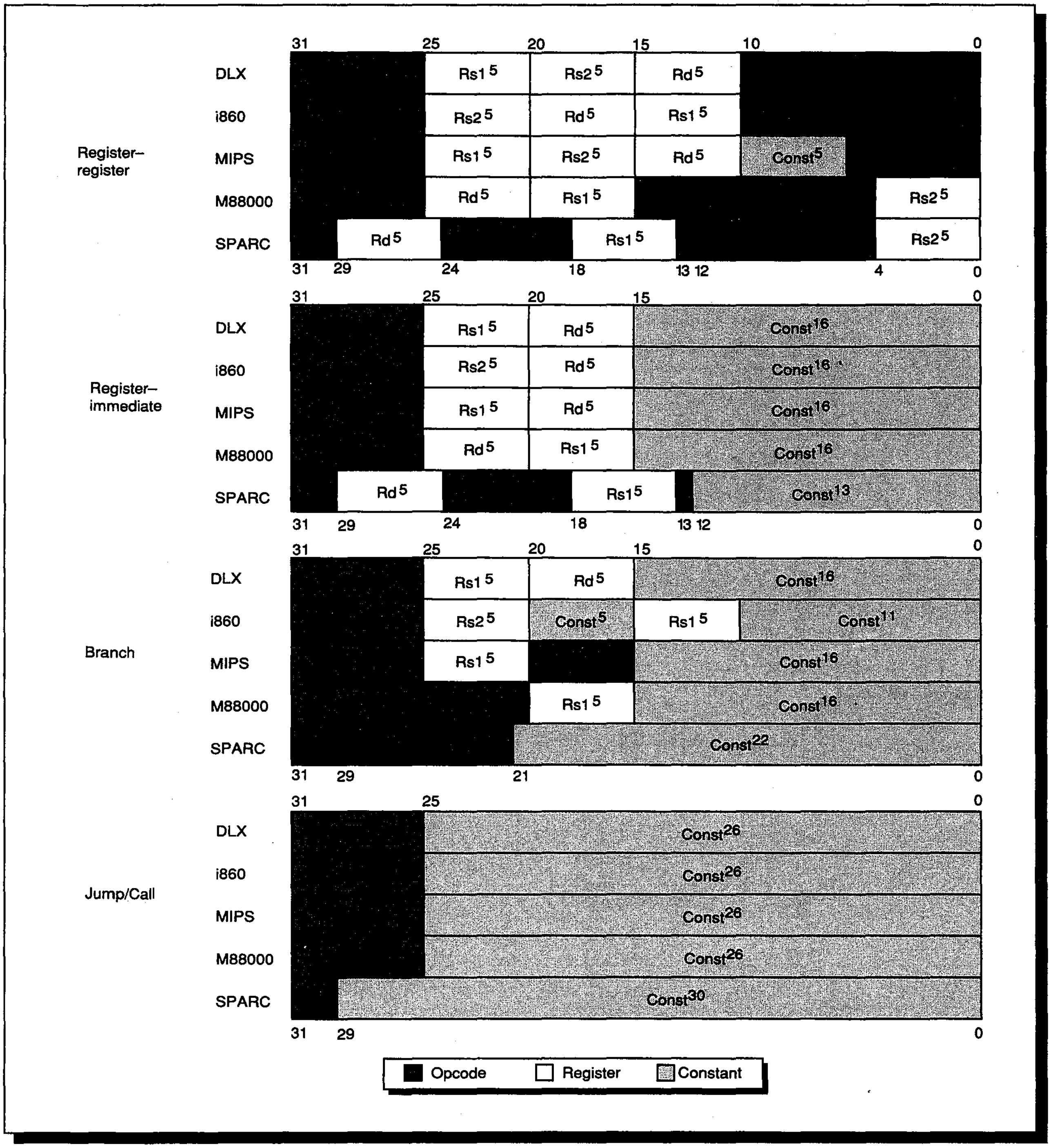


FIGURE E.3 Instruction formats for five architectures. These four formats are found in all five architectures. (The superscript notation in this figure means something different from our standard notation; it shows the width of a field in bits.) While the register fields are located in similar pieces of the instruction, beware that the destination and two source fields are scrambled. Here are the meanings of the abbreviations: Op = the main opcode, Opx =an opcode extension, Rd = the destination register, Rs1 = source register 1, Rs2 = source register 2, and Const = a constant (used as an immediate or as an address). The main variation for the M88000 is register–immediate format when the operation doesn't need a full 16-bit immediate: an opcode extension field is placed in the upper bits of the constant field. The variation for the i860 is using Rs1 in the Branch format to specify a 5-bit constant as well as a register.

References to code are normally PC-relative, although register indirect is supported for returning from procedures and for case statements. One variation is that PC-relative branch addresses in everything but DLX are shifted left 2 bits before being added to the PC, thereby increasing the branch distance. This works because the length of all instructions is one word and instructions must be word aligned in memory.

Figure E.3 (page E-3) shows the format of instructions, which includes the size of the address in the instructions. Each instruction set architecture uses these four primary instruction formats. The primary differences are subtle, concerning how to extend constant fields to 32 bits. Figure E.4 shows the variations.

Format: instruction category	**DLX**	**i860**	**MIPS**	**M88000**	**SPARC**
Branch: all	Sign	Sign	Sign	Sign	Sign
Jump/Call: all	Sign	Sign	--	Sign	Sign
Register–immediate: data transfer	Sign	Sign	Sign	Zero	Sign
Register–immediate: arithmetic	Sign	Sign	Sign	Zero	Sign
Register–immediate: logical	Sign	Zero	Zero	Zero	Sign

FIGURE E.4 Summary of constant extension. The constant in the Jump and Call instructions of MIPS are not sign extended since they only replace the lower 28 bits of the PC, leaving the upper 4 bits unchanged.

E.3 Instructions: The DLX Subset

The similarities of each architecture allow simultaneous descriptions of the architectures, starting with the operations equivalent to DLX.

DLX Instructions

Almost every instruction found in DLX instructions is found in the other architectures, as Figure E.5 shows. (For reference, definitions of the DLX instructions are found on pages 160 to 167 of Chapter 4 and the back inside cover.) Instructions are listed under four categories: "Data transfer," "Arithmetic, logical," "Control," and "Floating point." A fifth category in the figure shows conventions for register usage and pseudoinstructions on each architecture. If a DLX instruction requires a short sequence of instructions, these instructions are separated by semicolons in Figure E.5. (To avoid confusion, the destination register will **always** be the leftmost operand in this appendix, independent of the notation normally used with each architecture.)

Every architecture must have a scheme for compare and conditional branch, but even with all the similarities, each of these architectures has found a different way to perform the operation. The advantages and disadvantages of the general options are found on pages 105–109 of Chapter 3.

Instruction name	DLX	i860	MIPS	M88000	SPARC
Data transfer (Instruction formats)	**R–I**	**R–I, R–R**	**R–I**	**R–I, R–R**	**R–I, R–R**
Load byte signed	LB	LD.B	LB	LD.B	LDSB
Load byte unsigned	LBU	LD.B; AND ...,x00FF,...	LBU	LD.BU	LDUB
Load halfword signed	LH	LD.S	LH	LD.H	LDSH
Load halfword unsigned	LHU	LD.S; AND ...,xFFFF...	LHU	LD.HU	LDUH
Load word	LW	LD.L	LW	LD	LD
Load SP float	LF	FLD.L	LWC1	LD	LDF
Load DP float (see E.5 for MIPS)	LD	FLD.D	LWC1 Rd; LWC1 Rd+1	LD.D	LDDF
Store byte	SB	ST.B	SB	ST.B	STB
Store halfword	SH	ST.S	SH	ST.H	STH
Store word	SW	ST.L	SW	ST	ST
Store SP float	SF	FST.L	SWC1	ST	STF
Store DP float (see E.5 for MIPS)	SD	FST.D	SWC1 Rd; SWC1 Rd+1	ST.D	STDF
Read, write special registers	MOVS2I, MOVI2S	LD.C, ST.C	MF_, MT_	LDCR,FLDCR STCR,FSTCR	RD, LDFSR, WR, STFSR
Move int. to FP reg.	MOVI2FP	IXFR	MFC1	not applicable	ST;LDF,
Move FP to int. reg.	MOVFP2I	FXFR	MTC1	not applicable	STF;LD
Arithmetic, logical (Instruction formats)	**R–R, R–I**	**R–R, R–I**	**R–R, R–I**	**R–R, R–I**	**R–R, R–I**
Add	ADDU,ADDUI	ADD,ADDU	ADDU,ADDIU	ADDU	ADD
Add (trap if overflow)	ADD,ADDI	ADD; INTOVR	ADD,ADDI	ADD	ADDcc; TVS
Sub	SUBU,SUBUI	SUB,SUBU	SUBU	SUBU	SUB
Sub (trap if overflow)	SUB,SUBI	SUB; INTOVR	SUB	SUB	SUBcc; TVS
Multiply (see E.6 for SPARC)	MULTU, MULTUI	FMLOW	MULT, MULTU	MUL	MULScc;....; MULScc
Multiply (trap if ovf)	MULT,MULTI	--	--	--	-- (see E.6)
Divide	DIVU,DIVUI	--	DIV,DIVU	DIV,DIVU	-- (see E.6)
Divide (trap if ovf)	DIV,DIVI	--	--	--	-- (see E.6)
And	AND,ANDI	AND	AND,ANDI	AND	AND
Or	OR,ORI	OR	OR,ORI	OR	OR
Xor	XOR,XORI	XOR	XOR,XORI	XOR	XOR
Load high part reg.	LHI	OR.H ...,r0,...	LUI	OR.U ...,r0,...	SETHI *(B fmt.)*
Shift left logical	SLL,SLLI	SHL	SLLV,SLL	MAK	SLL
Shift right logical	SRL,SRLI	SHR	SRLV,SRL	EXTU	SRL
Shift right arithmetic	SRA,SRAI	SHRA	SRAV,SRA	EXT	SRA
Compare	S-(<,>,≤,≥,=,≠)	SUB r0,...	SLT,SLTU, SLTI,SLTIU	CMP	SUBcc r0,...

Instruction Name	DLX	i860	MIPS	M88000	SPARC
Control (Instruction formats)	**B, J/C**	**B, J/C**	**B, J/C**	**B, J/C**	**B, J/C**
Branch on integer compare	BEQ,BNE	BC.T,BNC.T, BTE,BTNE	BEQ,BNE,B_Z (<,>,≤,≥)	BB1.N,BB0.N, BCND.N	Bicc (<,>,≤,≥,=,≠)
Branch on floating-point compare	BFPT,BFPF	BC.T,BNC.T	BC1T,BC1F	BB1.N,BB0.N, BCND.N	FBfcc (<,>,≤,≥,=,…)
Jump, jump register	J,JR	BR, BRI	J,JR	BR.N,JMP.N	B, JMPL r0,...
Call, call register	JAL,JALR	CALL, CALLI	JAL,JALR	BSR.N,JSR.N	CALL,JMPL
Trap	TRAP	TRAP	BREAK	TCND, TB0	Ticc
Return from interrupt	RFE	BRI *(trap bits≠0)*	JR; RFE	RTE	RETT
Floating point (Instruction formats)	**R–R**	**R–R**	**R–R**	**R–R**	**R–R**
Add single, double	ADDF, ADDD	FADD.SS, FADD.DD	ADD.S, ADD.D	FADD.SSS, FADD.DDD	FADDS, FADDD
Sub single, double	SUBF, SUBD	FSUB.SS, FSUB.DD	SUB.S, SUB.D	FSUB.SSS, FSUB.DDD	FSUBS, FSUBD
Mult single, double	MULF, MULD	FMUL.SS, FMUL.DD	MUL.S, MUL.D	FMUL.SSS, FMUL.DDD	FMULS, FMULD
Div single, double	DIVF, DIVD	--, --	DIV.S, DIV.D	FDIV.SSS, FDIV.DDD	FDIVS, FDIVD
Compare	_F, _D (<,>,≤,≥,=,…)	PF_.SS, PF_.DD (>,≤,=)	C_.S, C_.D (<,>,≤,≥,=,…)	FCMP.SS, FCMP.DD	FCMPS, FCMPD
Move R–R	MOVF	FIADD.SS ...,f0,	MOV.S	ADD ...,r0,...	FMOVS
Convert (single,double,integer) to (single,double,integer)	CVTF2D, CVTD2F, CVTF2I, CVTD2I, CVTI2F, CVTI2D	FADD.SD ..f0.., FADD.DS ..f0.., FIX.SS, FIX.DS, --, --	CVT.S.D, CVT.D.S, CVT.S.W, CVT.D.W, CVT.W.S, CVT.W.D	FADD.SSD r0, --, INT.SS, INT.SD, FLT.SS, FLT.DS	FSTOD, FDTOS, FSTOI, FDTOI, FITOS, FITOD
Conventions					
Register with value 0	r0	r0	r0	r0	r0
Return address reg.	r31	r1	r31	r1	r31
Noop	ADD r0,r0,r0	SHL r0,r0,r0	SLL r0,r0,r0	OR r0,r0,r0	SETHI r0,0
Move R–R integer	ADD ...,r0,...	SHL ...,r0,...	ADD ...,r0,...	OR ...,r0,...	OR ...,r0,...
Operand order	OP Rd,Rs1,Rs2	OP Rs1,Rs2,Rd	OP Rd,Rs1,Rs2	OP Rd,Rs1,Rs2	OP Rs1,Rs2,Rd

FIGURE E.5 Instructions equivalent to DLX. Dashes mean the operation is not available in that architecture, or not synthesized in a few instructions. Such a sequence of instructions is shown separated by semicolons. If there are several choices of instructions equivalent to DLX, they are separated by commas. Finally, "not applicable" means that while this operation is not directly available, other changes in the architecture means it wouldn't make sense. This later category is for the M88000, since integer and floating-point instructions sharing the same registers means separate floating-point move instructions are unnecessary. Note that in the "Arithmetic, logical" category DLX and MIPS use separate instruction mnemonics to indicate an immediate operand, while the i860, M88000, and SPARC offer immediate versions of these instructions but use a single mnemonic. (Of course these are separate opcodes!) Both MIPS and SPARC have new instructions that were not implemented in the first machine and that apply to some of these cases: see Sections E.5 and E.6.

Compare and Conditional Branch

SPARC uses the traditional four condition code bits stored in the program status word: Negative, Zero, Carry, and Overflow. They can be set on any arithmetic or logical instruction, but unlike earlier architectures this setting is optional on each instruction. This leads to fewer problems in pipelined implementation (page 334 in Chapter 6). While condition codes can be set as a side effect of an operation, explicit compares are synthesized with a subtract using r0 as the destination. Floating point uses separate condition codes to encode the IEEE 754 conditions, requiring a floating-point compare instruction. SPARC conditional branches test condition codes to determine all possible unsigned and signed relations.

MIPS uses the contents of registers to evaluate conditional branches. Any two registers can be compared for equality (BEQ) or inequality (BNE) and then the branch is taken if the condition holds. The set-on-less-than instructions (SLT,SLTI, SLTU,SLTIU) compare two operands and then set the destination register to 1 if less and to 0 otherwise. These instructions are enough to synthesize the full set of relations. Because of the popularity of comparisons to 0, MIPS includes special compare-and-branch instructions for all such comparisons: greater than or equal to zero (BGEZ), greater than zero (BGTZ), less than or equal to zero (BLEZ), and less than zero (BLTZ). Of course, equal and not equal to zero can be synthesized using r0 with BEQ and BNE. Like SPARC, MIPS uses a condition code for floating point with separate floating-point compare and branch instructions.

The M88000 also uses registers to evaluate conditions and optimizes compare to 0 with a separate set of compare-and-branch instructions (BCND.N). Comparison of arbitrary operands differs. MIPS offers several compare instructions to set the register to 0 or 1 depending on the selected condition, but the M88000 uses a single instruction (CMP) and sets 10 bits of the destination register showing the relationship of the two operands. These bits represent equality (=, ≠) plus all relations for signed (<, ≤, >, ≥) and unsigned (<, ≤, >, ≥) operands. Instructions that branch if a bit in a register is 1 (BB1.N) or 0 (BB0.N) complete the conditional branch set. (Another option is using EXTU with CMP to set a register to 0 or 1 and then using BCND.N. Using EXT instead of EXTU sets a register to 0 or –1, if so desired.) Since there is a common register set for integer and floating point, floating-point compare uses the same scheme: set bits of a register and branch based on the result using BB1.N or BB0.N.

The Intel i860 uses condition codes for branches like SPARC, except that the i860 condition codes are set implicitly as part of every integer arithmetic or logical instruction. Also unlike SPARC, the i860 uses just two bits of conditions: OF and CC. OF is set only by the integer add and subtract instructions, and is used to indicate overflow. There is no conditional branch instruction to test this bit, but the INTOVR instruction will cause a trap if the bit is set. The CC bit is set or cleared depending on the operation. The logical instructions (AND,OR,XOR) set CC if the result is 0. The unsigned arithmetic instructions (ADDU,SUBU) set CC if there is a carry out of the most significant bit. Signed subtract (SUBS) sets CC if Rs2 > Rs1, while signed add (ADDS) sets CC if Rs2 is less than the two's complement of Rs1. Floating-point comparison instructions set CC if the condition tested is true: greater than (PFGT), less than or equal (PFLE), or equal (PFEQ).

The i860 conditional branch instructions (BC.T and BNC.T) test CC and branch depending on whether CC is 1 or 0. The i860 also has conditional branch instructions based on equality of two operands: BTE jumps if they are equal and BTNE jumps if they are not.

Figure E.6 summarizes the four schemes used for conditional branches.

	DLX	i860	MIPS	M88000	SPARC
Number of condition code bits (integer and FP)	1 FP	1 both, 1 integer	1 FP	--	4 integer, 2 FP
Basic compare instructions (integer and FP)	1 integer, 1 FP	1 FP	1 integer, 1 FP	1 integer, 1 FP	1 FP
Basic branch instructions (integer and FP)	1 integer, 1 FP	1 both, 1 integer	2 integer, 1 FP	1 both, 1 integer	1 integer, 1 FP
Compare register with register/const and branch	=,≠	=,≠	=,≠	--	--
Compare register to zero and branch	=,≠	=,≠	=,≠,<,≤,>,≥	=,≠,<,≤,>,≥	--

FIGURE E.6 Summary of five approaches to conditional branches. Integer compare on the i860 and SPARC is synthesized with an arithmetic instruction that sets the condition codes using r0 as the destination.

Integer Multiply and Divide

Multiply and divide are usually implemented as multicycle instructions and are thus not a good match for the single-cycle execution goal of the rest of the integer instructions, requiring separate integration into the pipeline. Each architecture takes a different approach to integer multiply and divide as well as conditional branch. The i860 uses the same scheme as DLX: there is a floating-point instruction (FMLOW) that treats the contents of two floating-point registers as integers, leaving a 32-bit result in the lower 32 bits of a double-precision pair of floating-point registers. Programs do integer divide using i860 floating-point instructions. (Floating-point divide uses Newton-Raphson iteration; see pages E-19–E-20.)

The combined integer and floating-point register file allows the M88000 to use the floating-point unit to perform integer multiply and divide, as the operands do not have to be moved to and from the floating-point registers. The one complication in the first version of the architecture, the MC88100, is a negative dividend or negative divisor results in a trap. Software then makes the operands positive, uses the divide instruction, and then complements the quotient (if necessary). A zero divisor traps as well, as we would hope.

In the MIPS architecture the 64-bit product of an integer multiply or the quotient/remainder of an integer divide is placed in a special registers HI and LO. This computation is treated as an independent unit executing in parallel with the integer and floating-point units. The appropriate result is transferred to the correct register with a MFHI or MFLO instruction. Attempts to read the registers before the computation is complete stalls the processor. There is no trap for overflow or divide by zero. These are typically checked by explicit integer instructions that execute in parallel with the divide. (See Section E.5 for architectural extensions not implemented in the first MIPS machines.)

SPARC provides a multiply step instruction. When used in a loop it calculates a full 64-bit product using the special register Y. It is loaded with the multiplier and receives the least significant word of the product. Magenheimer, Peters, Pettis, and Zuras [1988] measured the size of operands in multiplies and divides to show how well the multiply step would work. Using this data for C programs, Muchnick [1988] found that by making special cases the average multiply by a constant takes 6 clock cycles and multiply of variables takes 24 clock cycles. There is no divide step in the SPARC. (See Section E.6 for architectural extensions not implemented in the first SPARC machines.)

E.4 Instructions: Common Extensions to DLX

Figure E.7 (pages E-10–E-11) lists instructions not found in Figure E.5 (pages E-5–E-6) in the same four categories. Instructions are put in this list if they appear in more than one of the four architectures. The instructions are defined using the hardware description language, which is described on the page facing the inside back cover and on pages 160–167 of Chapter 4.

While most of the categories are self-explanatory, a few bear comment:

- The “Atomic swap” row means a primitive that can exchange a register with memory without interruption. This is useful for operating system semaphores in uniprocessors as well as for multiprocessor synchronization (see pages 471–473 of Chapter 8.)
- In the “Endian” row, “Big or Little” means there is a bit in the program status register that allows the processor to act either as Big Endian or Little Endian. This can be accomplished by simply complementing some of the least significant bits of the address in data transfer instructions.
- The “Coprocessor operations” row lists several categories that allow for the processor to be extended with special-purpose hardware.
- The “Implicit conversions” row under “Floating point” means that floating-point operands in these architectures do not have to all be the same size, and the floating-point unit performs a conversion as part of the operation. The i860 allows for two single-precision operands to produce a double-precision

result while the M88000 allows for any combination of single and double precisions for each of the three operands.

One difference that needs a longer explanation is the optimized branches. Figure E.8 (page E-12) shows the options. The i860 and M88000 offer branches that take effect immediately, like branches on earlier architectures. This avoids executing NOPs when there is no instruction to fill the delay slot. SPARC provides a version of delayed branch that makes it easier to fill the delay slot. The "annulling" branch executes the instruction in the delay slot only if the branch is taken; otherwise the instruction is annulled. This means the instruction at the target of the branch can safely be copied into the delay slot since it will only be executed if the branch is taken. The restrictions are that the target is not another branch and that the target is known at compile time. SPARC also offers a nondelayed jump because an unconditional branch with the annul bit set does **not** execute the following instruction.

After covering the similarities, we will cover the unique features of each architecture, ordering them by length of description of the unique features from shortest to longest.

Name	**Definition**	**i860**	**MIPS**	**M88000**	**SPARC**
Data transfer					

Atomic swap R/M (for semaphores)	Temp←Rd;
	Rd← Mem[x];
	Mem[x]←Temp
		LOCK;LD.L;
		UNLOCK; ST.L;
			-- (see E.5)	XMEM,
				XMEMBU
					SWAP

Load double integer	Rd←Mem[x]; Rd+1←Mem[x+4]	--	--	LD.D	LDD
Store double integer	Mem[x]←Rd; Mem[x+4]←Rd+1	--	--	ST.D	STD
Load coprocessor	Coprocessor←Mem[x]	--	LWCi	--	LDC
Store coprocessor	Mem[x]←Coprocessor	--	SWCi	--	STC
Endian	(Big/Little Endian?)	Big or Little	Big or Little	Big or Little	Big
Cache flush	(Flush cache block at this address)	FLUSH	-- (see E.5)	--	FLUSH
Arithmetic, logical					
Support for multi-word integer add	CarryOut,Rd ← Rs1 + Rs2 + OldCarryOut	ADDU;BNC; ADDU ...,...,#1	ADDU;SLTU; ADDU	ADDU.CIO	ADDXcc
Support for multi-word integer sub	CarryOut,Rd ← Rs1 – Rs2 + OldCarryOut	SUBU;BNC; ADDU ....,...,#1	SUBU;SLTU; SUBU	SUBU.CIO	SUBXcc
And not	Rd ← Rs1 & !(Rs2)	ANDNOT	--	AND.C *(R–R)*	ANDN
Or not	Rd ← Rs1 | !(Rs2)	--	--	OR.C *(R–R)*	ORN
Xor not	Rd ← Rs1 ^ !(Rs2)	--	--	XOR.C *(R–R)*	XNOR

	Definition	**i860**	**MIPS**	**M88000**	**SPARC**
Arithmetic, logical *(continued)*					
And high immediate	$Rd_{0..15} \leftarrow Rs1_{0..15}$ & (Const<<16); $Rd_{16..31} \leftarrow 0$	`ANDH` *(R–I)*	--	`AND.U` *(R–I)*	--
Or high immediate	$Rd_{0..15} \leftarrow Rs1_{0..15} \mid$ (Const<<16); $Rd_{16..31} \leftarrow 0$	`ORH` *(R–I)*	--	`OR.U` *(R–I)*	--
Xor high immediate	$Rd_{0..15} \leftarrow Rs1_{0..15}$ ^ (Const<<16); $Rd_{16..31} \leftarrow 0$	`XORH` *(R–I)*	--	`XOR.U` *(R–I)*	--
Coprocessor operations	(Defined by coprocessor)	--	`COPi`	--	`CPop`
Control					

Optimized delayed branches	(Branch not always delayed)	`BC,BNC`	--	`BB1,BB0,`
				`BCND`
					`Bicc,A`

Optimized floating-point branches	(Branch not always delayed)	`BC,BNC`	--	`BB1,BB0,`
				`BCND`
					`Bfcc,A`

Conditional trap	if (COND) {R31←PC; PC ←0..0#i}	--	-- (see E.5)	`TB1, TB0,`
				`TCND`
					`Ticc`

Branch on coprocessor	if (CoProc COND) {PC ←PC+Const}	--	`BCiT,BCiF`	--	`Bccc`
No. control regs.	Misc. regs (virtual memory, interrupts,...)	6	12	32	7
Floating point					

Negate	Fd ← Fs ^ x80000000	--	`NEG.S,`
			`NEG.D`
				`XOR.U 8000`	`NEGS`

Absolute value	Fd ← Fs & x7FFFFFFF	--	`ABS.S,`
			`ABS.D`
				`AND.U 7FFF`	`ABSS`

Truncate to integer	Fd ← unrounded integer part of Fs	`FTRUNC.SS,`
		`FTRUNC.DS`
			--	`TRNC.SS,`
				`TRNC.SD`
					--

Implicit conversions	(Convert as part of operation)	`_.SD`
		(2 single operands, 1 double result)
			--	`_.SSD,_.SDS,`
				`_.SDD,_.DSS,`
				`_.DSD._.DDS`
				(all combinations)
					--

FIGURE E.7 Instructions not found in DLX but found in two or more of the four architectures. Both MIPS and SPARC have new instructions that were not implemented in the first machine and that apply to some of these cases: see Sections E.5 and E.6.

	Delayed branch	(Plain) Branch	Annulling delayed branch
Found in architectures	All 5 RISCs	i860, M88000	SPARC
Execute following instruction	Always	Only if branch **not** taken	Only if branch taken

FIGURE E.8 When the instruction following the branch is executed for three types of branches.

E.5 Instructions Unique to MIPS

Starting with data transfer instructions, MIPS is unlike the others since the architecture requires that the instruction following a load does not refer to the value being loaded. The MIPS Assembler inserts a NOOP instruction if this situation occurs.

Nonaligned Data Transfers

The other unique feature of MIPS data transfer is special instructions to handle misaligned words in memory. A rare event in most programs, it is included for COBOL programs where the programmer can force misalignment by declarations. While all these architectures trap if you try to load a word or store a word to a misaligned address, on all architectures misaligned words can be accessed without traps by using 4 load byte instructions and then assembling the result using shifts and logical ORs. The MIPS load and store word left and right instructions (`LWL`, `LWR`, `SWL`, `SWR`) allow this to be done in just 2 instructions: `LWL` loads the left portion of the register and `LWR` loads the right portion of the register. `SWL` and `SWR` do the corresponding stores. Figure E.9 shows how they work. Unlike other loads, a `LWL` followed by a `LWR` does not require a NOOP even though both will specify the same register since fields do not overlap.

TLB Instructions

TLB misses are handled in software in the MIPS R2000, so the instruction set also has instructions for manipulating the registers of the TLB (see pages 437–438 and 443–445 in Chapter 8 for more on TLBs.) These registers are considered part of the "system coprocessor" and thus can be accessed by the instructions that move between coprocessor registers and integer registers. The contents of a TLB entry are read by loading via Read Indexed TLB Entry (`TLBR`) and written using either Write Indexed TLB Entry (`TLBWI`) or Write Random TLB Entry (`TLBWR`). The TLB contents are searched using Probe TLB for Matching Entry (`TLBP`).

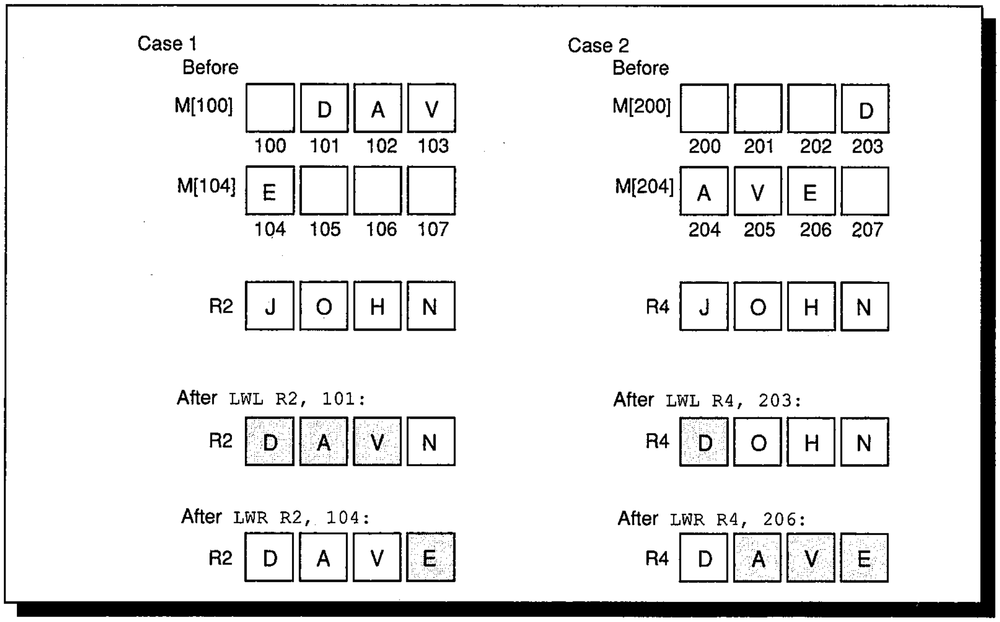


FIGURE E.9 MIPS instructions for unaligned word reads. This figure assumes operating in Big Endian mode. Case (1) first loads the 3 bytes 101,102, and 103 into the left of R2 leaving the least significant byte undisturbed. The following LWR simply loads byte 104 into the least significant byte of R2 leaving the other bytes of the register unchanged using LWL. Case (2) first loads byte 203 into the most significant byte of R4 and the following LWR loads the other 3 bytes of R4 from memory bytes 204, 205, and 206. LWL reads the word with the first byte from memory, shifts to the left to discard the unneeded byte(s), and changes only those bytes in Rd. The byte(s) transferred are from the first byte until the lowest-order byte of the word. The following LWR addresses the last byte, right shifts to discard the unneeded byte(s), and finally changes only those bytes of Rd. The byte(s) transferred are from the last byte up to the highest-order byte of the word. Store word left (SWL) is simply the inverse of LWL, and store word right (SWR) is the inverse of LWR. Changing to Little Endian mode flips which bytes are selected and discarded. (If big/little–left/right–load/store seems confusing, don't worry, it works!)

Remaining Instructions

Below is a list of the remaining unique details of the MIPS architecture:

- *NOR:* This logical instruction calculates !(Rs1 | Rs2).
- *Constant shift amount:* Nonvariable shifts use the 5-bit constant field shown in the register–register format in Figure E.3.
- *SYSCALL:* This special trap instruction is used to invoke the operating system.
- *Move to/from control registers:* CTCi and CFCi move between the integer registers and control registers.

- *Limited single-precision registers:* Although the 32 floating-point registers can be addressed individually for loads and stores, single-precision operands for floating-point operations can use only the 16 even floating-point registers.
- *Jump/Call not PC-relative:* The 26-bit address of jumps and calls is not added to the PC. It is shifted left 2 bits and replaces the lower 28 bits of the PC. This would only make a difference if the program was located near a 256-MB boundary.
- *Conditional procedure call instructions*: `BGEZAL` saves the return address and branches if the contents of Rs1 is greater than or equal to zero, and `BLTZAL` does the same for less than zero. The purpose of these instructions is to get a PC-relative call.

There is no specific provision in the MIPS architecture for floating-point execution to proceed in parallel with integer execution, but the MIPS implementations of floating point allow this to happen by checking to see if arithmetic interrupts are possible early in the cycle; normally interrupts are not possible and integer and floating point operate in parallel (see page A-31 in Appendix A).

MIPS II

With the announcement of the R6000 came a set of extensions to the original MIPS architecture described above. Here are the additions of MIPS II:

- *Interlocked loads:* The MIPS II Assembler need not insert a NOP after a load if there is a dependency on the following instruction, as the hardware will automatically stall.
- *Branch likely:* Equivalent to the SPARC annulled branches, this instruction executes the instruction in the delay slot only if the branch is taken.
- *Load double floating point and store double floating point:* MIPS II takes a single instruction to load or store double-precision floating-point numbers.
- *`SQRT`:* Single- and double-precision floating-point square root are added to the floating-point operations.
- *Conditional trap instructions:* These match the conditional branch instructions, except they are not delayed: When the trap is taken, the following instruction is **not** executed. These instructions are useful for range checking, popular in Ada.

E.6 Instructions Unique to SPARC

Register Windows

The primary unique feature of SPARC is register windows (pages 450–453 of Chapter 8), used to reduce the register save/restore overhead of procedure calls and returns. SPARC can have between 2 and 32 windows, using 8 registers each for the globals, locals, incoming parameters, and outgoing parameters (see Figure 8.34 page 452.) (Given each window has 16 unique registers, an implementation of SPARC can have as few as 40 physical registers and as many as 520, although most have 128 to 136, so far.) Rather than tie window changes with call and return instructions, SPARC has the separate instructions `SAVE` and `RESTORE`. `SAVE` is used to "save" the caller's window by pointing to the next window of registers in addition to performing an add instruction. The trick is that the source registers are from the caller's window of the addition operation while the destination register is in the callee's window. SPARC compilers typically use this instruction for changing the stack pointer to allocate local variables in a new stack frame. `RESTORE` is the inverse of `SAVE`, bringing back the caller's window while acting as an add instruction, with the source registers from the callee's window and the destination register in the caller's window. This automatically deallocates the stack frame. Compilers can also make use of it for generating the callee's final return value. Unlike earlier register window architectures, SPARC uses a Window Invalid Mask, which is used in real-time applications, that allows the windows to be partitioned between different processes.

Another data transfer feature is alternate space option for loads and stores. This simply allows the memory system to identify memory accesses to input/output devices, or to control registers for devices such as the cache and memory-management unit.

Support for LISP and Smalltalk

The primary remaining arithmetic feature is tagged addition and subtraction. The designers of SPARC spent some time thinking about languages like LISP and Smalltalk, and this influenced some of the features of SPARC already discussed: register windows, conditional trap instructions, calls with 32-bit instruction addresses, and multiword arithmetic (see Taylor [1986] and Ungar [1984]). A small amount of support is offered for tagged data types with operations for addition, subtraction, and hence comparison. The two least significant bits indicate whether the operand is an integer (coded as 00), so `TADDcc` and `TSUBcc` set the overflow bit if either operand is not tagged as integer or if the result is too large. A subsequent conditional branch or trap instruction can decide what to do. (If the operands are not integers, software recovers the operands, checks the types of the operands, and invokes the correct operation based on those types.) Two other versions of these instructions make the conditional trap unnecessary, as TADDccTV and TSUBccTV trap if the overflow is set. It turns out that the misaligned memory access trap can also be put to use for tagged data, since loading from a pointer with the wrong tag can be an invalid access. Figure E.10 shows both types of tag support.

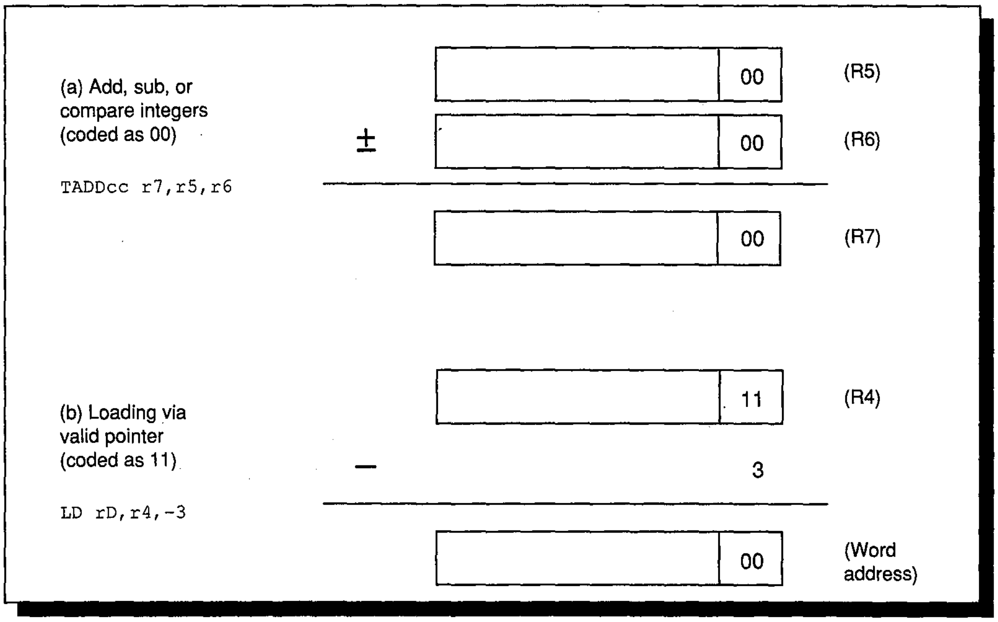

(a) Add, sub, or compare integers (coded as 00)
TADDcc r7,r5,r6
±
00
(R5)
00
(R6)
00
(R7)
(b) Loading via valid pointer (coded as 11)
LD rD,r4,-3
–
11
(R4)
3
00
(Word address)

FIGURE E.10 SPARC uses the two least significant bits to encode different data types for the tagged arithmetic instructions. (a) shows integer arithmetic, which takes a single cycle as long as the operands and the result are integers. (b) shows that the misaligned trap can be used to catch invalid memory accesses, such as trying to use an integer as a pointer. For languages with paired data like LISP, an offset of –3 can be used to access the even word of a pair (CAR) and +1 can be used for the odd word of a pair (CDR).

Overlapped Integer and Floating-Point Operations

SPARC allows floating-point instructions to overlap execution with integer instructions. To recover from an interrupt during such a situation, SPARC has a queue of pending floating-point instructions and their addresses. STDFQ allows the processor to empty the queue. The second floating-point feature is the inclusion of floating-point square root instructions FSQRTS and FSQRTD.

Remaining Instructions

The remaining unique features of SPARC are:

- *JMPL* uses Rd to specify the return address register, so specifying r31 makes it similar to JALR in DLX and specifying r0 makes it like JR.

- *LDSTUB* loads the value of the byte into Rd and then stores FF_{16} into the addressed byte. This instruction can be used to implement a semaphore.
- *LDDC and STDC* provide load double and store double for the coprocessor.
- *UNIMP* causes an unimplemented instruction interrupt. Muchnick [1988] explains how this is used for proper execution of aggregate returning procedures in C.

Finally, SPARC includes opcodes for instructions that are emulated in software on early implementations. SPARC application programs generally call dynamically linked library routines to perform these operations, but the opcodes would result in a trap if executed. The instructions are:

- *Signed and unsigned integer multiply and divide*, with both operands and the results being integer registers. The extra 32 bits of a product and the 32-bit remainder of a divide are placed in the Y register.
- *Quadruple precision floating-point arithmetic*, allowing the floating-point registers to act as eight 128-bit registers.
- *Multiple precision floating-point results for multiply*, meaning two single-precision operands can result in a double-precision product and two double-precision operands can result in a quadruple-precision product. These instructions can be useful in complex arithmetic and some models of floating-point calculations.

E.7 Instructions Unique to M88000

The most distinguishing feature of the M88000 is the single set of 32 registers for both integer and floating-point operations. This simplifies the instruction set at the cost of fewer registers for floating-point programs.

Bit Instructions

The next feature unique to the M88000 is a full set of bit-field instructions, shown in Figure E.11 (page E-18). (While we usually number the most significant bit 0, in this table we follow Motorola's notation, which numbers the most significant bit 31 and the least significant bit 0.) Bit-field instructions need an extra operand to specify the width of the field in addition to the destination register, source register, and beginning of the bit field. This 5-bit width field is located next to the bit field in source 2. The M88000 encodes a width of 0 to mean the full 32-bit value, hence the traditional shift instructions (SLL, SRL, SRA) are simply the corresponding bit-field instructions (MAK, EXTU, EXT) with 0 in the width field.

Name	Instruction	Notation
CLR	Clear bit field	$\texttt{Rd}_{(o+w)..(o+1)} \leftarrow 0^{w}$
SET	Set bit field	$\texttt{Rd}_{(o+w)..(o+1)} \leftarrow 1^{w}$
EXT	Extract signed bit field	$\texttt{if (w==0) \{Rd} \leftarrow \texttt{Rs1}_{31}{}^{o}\ \texttt{\#\# (Rs1 >> o) \}}$ $\texttt{else \{Rd} \leftarrow (\texttt{Rs1}_{(o+w)})^{o}\ \texttt{\#\#}\ (\texttt{Rs1}_{(o+w)..(o+1)}\ \texttt{>> o) \}}$
EXTU	Extract unsigned bit field	$\texttt{if (w==0) \{Rd} \leftarrow 0^{o}\ \texttt{\#\# (Rs1 >> o) \}}$ $\texttt{else \{Rd} \leftarrow 0^{o}\ \texttt{\#\#}\ (\texttt{Rs1}_{(o+w)..(o+1)}\ \texttt{>> o) \}}$
MAK	Make bit field	$\texttt{if (w==0) \{Rd} \leftarrow \texttt{Rs1 << o\}}$ $\texttt{else \{Rd}_{(o+w)..(o+1)} \leftarrow \texttt{Rs1}_{(w-1)..0}\ \texttt{\}}$
ROT	Rotate right	$\texttt{Rd} \leftarrow \texttt{Rs1}_{(o-1)..0}\ \texttt{\#\#}\ \texttt{Rs1}_{31..o}$
FF0	Find first bit clear	$\texttt{for (i=31;Rs2}_{i}\texttt{==0}\vert\vert\texttt{i<0;i} \leftarrow \texttt{i-1);}$ /* loop until = 0*/ $\texttt{if(i<0) \{Rd} \leftarrow \texttt{32\} else \{Rd} \leftarrow \texttt{i\}}$
FF1	Find first bit set	$\texttt{for (i=31;Rs2}_{i}\texttt{==1}\vert\vert\texttt{i<0;i} \leftarrow \texttt{i-1);}$ /* loop until =1 */ $\texttt{if(i<0) \{Rd} \leftarrow \texttt{32\} else \{Rd} \leftarrow \texttt{i\}}$

FIGURE E.11 The M88000 bit-field instructions. The bit offset, o, is the least significant five bits of the second operand and the bit-field width, w, is the five bits next to the offset. The subscript notation specifies a bit field while the superscript notation means replicate the bit that many times. Note that in this table, bit 31 refers to the most significant bit, and 0 refers to the least significant bit.

Remaining Instructions

The final unique instructions are load address (LDA), MASK, round to nearest integer (NINT), trap on bounds (TBND), and exchange control register (XCR):

- LDA loads Rd with the effective address rather than the data in memory. The only time this is different from ADDU is for scaled addressing of nonbyte data.
- MASK is simply another case of logical AND immediate: This instruction clears the other half of the word while AND immediate leaves it undisturbed. Thus, ANDI in DLX is arguably closer to MASK than to AND immediate in the M88000.
- NINT differs from INT in that it rounds to the nearest integer no matter how the rounding modes are set (see Appendix A, pages A-16 to A-17).
- TBND traps if Rs1 > Rs2, treating them as unsigned numbers (see page 239 in Chapter 5 for an explanation of how an unsigned comparison can check two signed bounds at once).
- XCR exchanges a control register with an integer register.

In addition to instructions, here are a few features that distinguish the M88000:

- Double-length operations use Rn and Rn+1 rather than an even-odd register pair. This gives the M88000 more flexibility in register allocation, which is important given the lack of floating-point registers.
- The first implementation, the MC88100, allows all multicycle instructions to overlap execution with following instructions unless there is a data hazard (see pages 264–265 in Chapter 6). Also, all floating-point instructions except divide are pipelined, taking just one cycle to issue single-precision operations and two cycles to issue double-precision operations. The 88000 provides a set of shadow registers (see Section 5.6) for floating-point operands to help software handle both precise and imprecise interrupts (see Motorola [1988]).
- There are special data transfers, identified by appending .USR to the instructions, that allow access to the user's data while in supervisor mode.

E.8 Instructions Unique to i860

The i860 has many unique features. Before covering the special extensions for graphics and high-performance floating point, let's cover the traditional areas.

The unique data transfers are for floating point only. The i860 provides 128-bit loads (`FLD.Q`) and stores (`FST.Q`) of pairs of 64-bit floating-point registers. It also provides an optional addressing mode on all floating-point loads and stores: the effective address (sum of Rs1/Const and Rs2) is stored back into Rs2. One unique characteristic is that the i860 seems to run out of opcode bits for load instructions because it uses the least significant bit to distinguish load halfword from load word. This works fine for the register–register format since bit 0 is an opcode extension field in this format, but in register–immediate format this is the least significant bit of the constant field. To avoid crazy addressing problems, this bit is cleared when used as an address. This prevents having an odd value in an index register that is corrected by an odd byte address in the constant field for halfword and word data transfers (see E.10(b) on page E-16 for a reason this is useful.)

The only unique arithmetic logical instruction is a double-length shift-right logical (`SHRD`). Rs1 and Rs2 are shifted right as a pair and then the 32 least significant bits are placed into Rd. Since there is no room in the instruction to specify the shift amount, `SHRD` uses the shift amount from the last `SHR` instruction. This value is saved in the 5-bit SC field of the program status word. By the way, `SHRD` can be used to perform a 32-bit rotate by having Rs1 and Rs2 specify the same register.

The i860 control instructions include a loop instruction called `BLA`. This instruction both performs an add and a conditional branch. Since it is likely that another instruction in the loop would change the condition code, the i860 has a special loop condition code (`LCC`) just for this instruction. `BLA` performs `Rd` ← `Rs1+Rs2` and branches if `LCC` equals 1. In addition, `BLA` sets the `LCC` for the next time through the loop if Rs2 ≥ –Rs1 and clears it otherwise. (LCC is set just the opposite of how ADDS sets CC.)

While i860 does not have floating-point divide, it does have a floating-point reciprocal instruction (FRCP). Used with Newton-Raphson iteration (pages A-23–A-24 of Appendix A), this calculates divide that disagrees with the IEEE floating-point standard (IEEE 754) in the 2 least significant bits. Intel offers software to produce the correctly rounded result at twice the cycle count. A similar instruction, FRSQR, calculates a reciprocal step for square root. The floating-point instructions also include 64-bit integer addition and subtraction (FIADD.DD and FISUB.DD) using the floating-point registers.

This covers the unique features in the traditional categories, so let's describe the new categories of the i860.

Graphics Instructions

The graphics or *pixel* instructions of i860 operate on 64 bits of data at a time, with each word representing several pixels. Pixel instructions are intended to be useful in graphics operations such as hidden surface elimination (see page 525 in Chapter 9), distance interpolation, and three-dimensional shading using intensity interpolation. These special-purpose instructions are not simple to understand, so interested readers should refer to the manual for details.

The overview of the operations is that two bits in the program status word determine the size of the pixels in a 64-bit word. Pixels can be 8-, 16-, or 32-bits wide, with each size containing fields representing intensity of the primary colors red, blue, and green. Some pixel instructions work with a 64-bit accumulator called the MERGE register, useful in collecting the results of a series of calculations on pixels. In addition to "merge" instructions (FADDP and FADDZ), the i860 has instructions for z buffers (page 525) that compare two sets of four 16-bit (FZCHKS) or two 32-bit (FZCHKL) values, storing the smaller values in the 64-bit destination register and setting bits indicating which was smaller in the program status word. Pixel-store instructions (PST) then use those bits to selectively store only those pixels that were smaller. Finally, the FORM instruction is used to move the MERGE register into a floating-point register and then clear MERGE.

Pipelined Mode

For higher performance, the i860 offers pipelined versions of all the floating-point and pixel instructions. One model for these instructions is to use them to build vector primitives, allowing procedures to be written to implement vector operations (see Chapter 7). The hope is that existing vectorizing compilers could invoke these more efficient procedures. Another model, used by compilers currently under development at Intel's behest, tries to compile directly into these instructions for both vector and nonvector codes.

In *pipelined mode,* an instruction is launched every cycle, but unlike other pipelined machines, there is no hardware to remember where the results are to be stored. Basically, the instruction issuing at the stage the pipeline completes specifies the destination! There are four independent pipelines in the i860, and each pipeline advances only when the next instruction of that type is executed. Figure E.12 shows the i860 pipelines, the number of pipeline stages, and instructions that advance each pipeline. Thus, the source fields and opcode specify the operation to be launched while the destination field specifies the register to be loaded by an instruction of the same type that is in the final stage at this cycle.

Pipeline	No. of Stages	Instructions using pipeline
FP multiplier	3 (single operands) 2 (double operands)	PFMUL
FP adder	3	PFADD, PFSUB, PFGT, PFLE, PFEQ, PFIX, PFTRUNC
FP load	3	PFLD
Graphics	1	PFIADD, PFISUB, PFZCHKS, PFZCHKL, PFADDP, PFADDZ, PFORM

FIGURE E.12 i860 pipelines, including the number of pipeline stages and instructions. All adder and multiplier instructions allow single-precision operands with single-precision results (.SS), single operands with double results (.SD), and double-precision operands with double-precision results (.DD). Since the number of stages differs for multiply depending on single or double, Intel recommends not mixing precisions involving multiplication.

For example, look at the sequence below for the floating-point adder pipeline (assume the operands are specified with the result on the left):

```
PFADD.SS    F4, F2, F3       ;Single Prec. Add
PFSUB.DD    F10, F8, F6      ;Double Prec. Sub
PFMUL.DD    F16, F12, F14    ;Double Prec. Mul
PFADD.SS    F19, F17, F18    ;Single Prec. Add
PFADD.SS    F22, F20, F21    ;Single Prec. Add
```

The floating-point adder pipeline is three stages, so the first instruction launches a floating-point add of F2 and F3, but F4 is loaded from the operation in the adder pipeline launched three instructions earlier. The multiply in this sequence does not advance the adder pipeline, so the third adder pipeline instruction following the first instruction (one subtract and two adds) is the final instruction in the sequence, meaning that F22 ← F2+F3.

The load pipeline has an interesting interaction with the data cache. As long as the data is in the cache, it is fetched from the cache. On a miss the data is fetched from memory, but the cache is not updated with the new data. This policy prevents operations on large data structures from filling the cache with data that will not be reused and throwing out data that would be reused. The programmer must decide on whether to use scalar loads (FLD) or pipelined loads (PFLD), depending on whether the data is likely to be reused or not.

Scalar instructions will normally empty the pipeline. (The exception is the load pipeline because FLD or LD don't empty it.) Thus, before executing a scalar floating-point instruction there must be a sequence of dummy pipelined instructions that store the results away. For example, there is no pipelined version of the floating-point instruction used for integer multiply (FMLOW), so the pipeline must be drained if an integer multiply is needed during a floating-point calculation.

Summarizing pipelined mode on the i860, the advantages are

- Pipeline control is simple (basically it is done in software).
- It doesn't need many registers, since they are not reserved during the operation.

The disadvantages are:

- Operations must be performed to empty the pipeline.
- The interrupt mechanism is complicated, taking longer to recover the state.
- Sometimes the pipeline is hard to use.
- Code size may mushroom (this has not yet been quantified).

Add/Sub and Multiply

To squeeze even more performance from the floating-point unit, the i860 has pipelined instructions that simultaneously perform an add and multiply (PFAM and PFMAM) or a subtract and multiply (PFSM and PFMSM), advancing the pipelines of both the add and multiply units. Since each instruction needs 4 sources and 2 destinations, the i860 has three registers that can also be used in addition to the three floating-point registers specified in the instruction. The registers KI and KR, optionally loaded from Rs1, can be sources for the multiplier, and register T can be a destination of the multiplier or a source for the adder. The final stage of adder pipeline and multiplier pipeline can also be sources. Four bits in each instruction specify a variety of combinations of the operands and the operations.

Dual Instruction Mode

Finally, the i860 allows an integer and a floating-point instruction to be fetched and executed simultaneously. This long instruction word or superscalar form of operation (pages 318–322 in Chapter 6) is called *dual-instruction mode* in the i860. Simultaneous execution occurs in this mode when the upper instruction of an aligned doubleword is an integer instruction and the lower is a floating-point instruction with the "D" bit set (bit 9 = 1). Entering or exiting the mode is delayed: When the i860 finds an instruction with the D bit set, it executes one more instruction before entering dual-instruction mode; and, similarly, when the i860 is in dual-instruction mode and finds a D bit not set, it executes one more pair before going to sequential execution.

Clearly, highest performance comes when the i860 is in both dual-instruction and pipelined modes.

E.9 Concluding Remarks

This appendix covers the addressing modes, instruction formats, and all instructions found in four recent architectures. While the later sections concentrate on the differences, it would not be possible to cover four architectures in these few pages if there were not so many similarities. In fact, we would guess that more than 90% of the instructions executed for any of these architectures would be found in Figure E.3 (page E-3). To illustrate this homogeneity, Figure E.13 gives a summary for four architectures from the 1970s similar to Figure E.1 (page E-2). (Imagine trying to write a single appendix in this style for those architectures.) In the history of computing, there has never been such widespread agreement on computer architecture.

	IBM 360/370	Intel 8086	Motorola 68000	DEC VAX
Date announced	1964/1970	1978	1980	1977
Instruction size(s) (bits)	16, 32, 48	8,16,24,32, 40,48	16,32,48, 64,80	8,16,24,32, ..., 432
Addressing (size, model)	24 bits, flat	4+16 bits, segmented	24 bits, flat	32 bits, flat
Data aligned?	Yes 360/ No 370	No	16-bit aligned	No
Data addressing modes	4	5	9	≥ 14
Protection	Page	None	Optional	Page
Page size	4 KB	--	0.25 to 32 KB	0.5 KB
I/O	Opcode	Opcode	Memory mapped	Memory mapped
Integer registers (size, model, number)	16 GPR x 32 bits	8 dedicated data x 16 bits	8 data & 8 address x 32 bits	15 GPR x 32 bits
Separate floating-point registers	4 x 64 bits	Optional: 8 x 80 bits	Optional: 8 x 80 bits	0
Floating-point format	IBM	IEEE 754 single, double, extended	IEEE 754 single, double, extended	DEC

FIGURE E.13 Summary of four 1970s architectures. Unlike the architectures in Figure E.1 (page E-2), there is little agreement between these architectures in any category. (See Chapter 4 for more details on the 370, 8086, and VAX.)

This style of architectures cannot remain static, however. One hard lesson is that address space must grow, so the 32-bit size of all these architectures must expand for them to survive. In terms of their implementation, we expect all to offer superscalar execution of 2 to 4 instructions per cycle. The hardware technology will go beyond the current CMOS VLSI and ECL to BiCMOS, and possibly even Gallium Arsenide. Our guess is that all of them will grow beyond the current market of workstations and peripheral controllers to minicomputers, mainframes, and even supercomputers, with increasing numbers of processors per computer class.

E.10 References

[1989]. *i860 64-Bit Microprocessor Programmer's Reference Manual.*

KANE, G. [1988]. *MIPS RISC Architecture,* Prentice-Hall, Englewood Cliffs, N. J.

MOTOROLA [1988]. *MC88100 RISC Microprocessor User's Manual.*

MAGENHEIMER, D. J., L. PETERS, K. W. PETTIS AND D. ZURAS [1988]. "Integer multiplication and division on the HP Precision Architecture," *IEEE Trans. on Computers,* 37:8, 980–990.

MUCHNICK, S. S. [1988]. "Optimizing compilers for SPARC," *Sun Technology* (Summer) 1:3, 64–77.

SUN MICROSYSTEMS [1989]. *The SPARC Architectural Manual,* Version 8, Part No. 800-1399-09, August 25, 1989.

TAYLOR, G., P. HILFINGER, J. LARUS, D. PATTERSON, AND B. ZORN [1986]. "Evaluation of the SPUR LISP architecture," *Proc. 13th Symposium on Computer Architecture (*June), Tokyo.

UNGAR, D., R. BLAU, P. FOLEY, D. SAMPLES, AND D. PATTERSON [1984]. "Architecture of SOAR: Smalltalk on a RISC," *Proc. 11th Symposium on Computer Architecture* (June), Ann Arbor, Mich., 188–197.

References

The following is a compilation of all the references listed in the reference section of each chapter. The page number of where each reference appears in the book is in parentheses after the reference.

ADAMS, T. AND R. ZIMMERMAN [1989]. "An analysis of 8086 instruction set usage in MS DOS programs," *Proc. Third Symposium on Architectural Support for Programming Languages and Systems* (April) Boston, 152–161. (p. 188)

AGARWAL, A. [1987]. *Analysis of Cache Performance for Operating Systems and Multiprogramming*, Ph.D. Thesis, Stanford Univ., Tech. Rep. No. CSL-TR-87-332 (May). (p. 487)

AGARWAL, A., R. L. SITES, AND M. HOROWITZ [1986]. "ATUM: A new technique for capturing address traces using microcode," *Proc. 13th Annual Symposium on Computer Architecture* (June 2–5), Tokyo, Japan, 119–127. (p. 486)

AGERWALA, T. AND J. COCKE [1987]. "High performance reduced instruction set processors," IBM Tech. Rep. (March). (p. 340)

ALEXANDER, W. G. AND D. B. WORTMAN [1975]. "Static and dynamic characteristics of XPL programs," *Computer* 8:11 (November) 41–46. (pp. 130, 187)

ALLIANT COMPUTER SYSTEMS CORP. [1987]. *Alliant FX/Series: Product Summary* (June), Acton, Mass. (p. 395)

ALMASI, G. S. AND A. GOTTLIEB [1989]. *Highly Parallel Computing,* Benjamin/Cummings, Redwood City, Calif. (p. 589)

AMDAHL, G. M. [1967]. "Validity of the single processor approach to achieving large scale computing capabilities," *Proc. AFIPS Spring Joint Computer Conf.* 30, Atlantic City, N. J. (April) 483–485. (pp. 26, 588)

AMDAHL, G. M., G. A. BLAAUW, AND F. P. BROOKS, JR. [1964]. "Architecture of the IBM System/360," *IBM J. Research and Development* 8:2 (April) 87–101. (pp. 127, 186)

ANDERSON, D. W., F. J. SPARACIO, AND R. M. TOMASULO [1967]. "The IBM 360 Model 91: Machine philosophy and instruction handling," *IBM J. of Research and Development* 11:1 (January) 8–24. (p. 339)

ANDERSON, S. F., J. G. EARLE, R. E. GOLDSCHMIDT, AND D. M. POWERS [1967]. "The IBM System/360 Model 91: Floating-point execution unit," *IBM J. Research and Development* 11, 34–53. Reprinted in [Swartzlander 1980]. (p. A-59)

ANDREWS, G. R. AND F. B. SCHNEIDER [1983]. "Concept and notations for concurrent programming," *Computing Surveys* 15:1 (March) 3–43. (p. 590)

ANON ET AL. [1985]. "A measure of transaction processing power," Tandem Tech. Rep. TR 85.2. Also appeared in *Datamation*, April 1, 1985. (p. 511)

ARCHIBALD, J. AND J.-L. BAER [1986]. "Cache coherence protocols: Evaluation using a multiprocessor simulation model," *ACM Trans. on Computer Systems* 4:4 (November) 273–298. (p. 487)

ATANASOFF, J. V. [1940]. "Computing machine for the solution of large systems of linear equations," Internal Report, Iowa State University. (p. 24)

ATKINS, D. E. [1968]. "Higher-radix division using estimates of the divisor and partial remainders," *IEEE Trans. on Computers* C-17:10, 925–934. Reprinted in [Swartzlander 1980]. (p. A-60)

BAER, J.-L. AND E.-H. WANG [1988]. "On the inclusion property for multi-level cache hierarchies," *Proc. 15th Annual Symposium on Computer Architecture* (May–June), Honolulu, 73–80. (p. 487)

BAKOGLU, H. B., G. F. GROHOSKI, L. E. THATCHER, J. A. KAHLE, C. R. MOORE, D. P. TUTTLE, W. E. MAULE, W. R. HARDELL, D. A. HICKS, M. NGUYEN PHU, R. K. MONTOYE, W. T. GLOVER , AND S. DHAWAN [1989]. "IBM second-generation RISC machine organization," *Proc. Int'l Conf. on Computer Design, IEEE* (October) Rye, N.Y., 138–142. (p. 340)

BANERJEE, U. [1979]. *Speedup of Ordinary Programs*, Ph.D. Thesis, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign (October). (p. 395)

BARTON, R. S. [1961]. "A new approach to the functional design of a computer," *Proc. Western Joint Computer Conf.*, 393–396. (p. 127)

BASHE, C. J., L. R. JOHNSON, J. H. PALMER, AND E. W. PUGH [1986]. *IBM's Early Computers*, MIT Press, Cambridge, Mass. (p. 561)

BASHE, C. J., W. BUCHHOLZ, G .V. HAWKINS, J .L. INGRAM, AND N. ROCHESTER [1981]. "The architecture of IBM's early computers," *IBM J. of Research and Development* 25:5 (September) 363–375. (p. 561)

BATCHER, K. E. [1974]. "STARAN parallel processor system hardware," *Proc. AFIPS National Computer Conf.*, 405–410. (p. 590)

BELL , C. G. AND W. D. STRECKER [1976]. "Computer structures: What have we learned from the PDP-11?," *Proc. Third Annual Symposium on Computer Architecture* (January), Pittsburgh, Penn., 1–14. (p. 485)

BELL, C. G. [1984]. "The mini and micro industries," *IEEE Computer* 17:10 (October) 14–30. (p. 27)

BELL, C. G. [1985]. "Multis: A new class of multiprocessor computers," *Science* 228 (April 26) 462–467. (p. 589)

BELL, C. G. [1989]. "The future of high performance computers in science and engineering," *Comm. ACM* 32:9 (September) 1091–1101. (p. 590)

BELL, C. G. AND A. NEWELL, [1971]. *Computer Structures: Readings and Examples*, McGraw-Hill, New York. (p. A-58)

BELL, C. G., J. C. MUDGE, AND J. E. MCNAMARA [1978]. *A DEC View of Computer Engineering*, Digital Press, Bedford, Mass. (p. 80)

BELL, C. G., R. CADY, H. MCFARLAND, B. DELAGI, J. O'LAUGHLIN, R. NOONAN, AND W. WULF [1970]. "A new architecture for mini-computers: The DEC PDP-11," *Proc. AFIPS SJCC*, 657–675. (p. 127)

BERRY, M., D. CHEN, P. KOSS, D. KUCK [1988]. "The Perfect Club benchmarks: Effective performance evaluation of supercomputers," CSRD Report No. 827 (November), Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign.'(p. 80)

BIRMAN, M., G. CHU, L. HU, J. MCLEOD, N. BEDARD, F. WARE, L. TORBAN, AND C. M. LIM [1988]. "Design of a high-speed arithmetic datapath," *Proc. ICCD: VLSI Computers and Processors*, 214–216. (p. A-53)

BLAKKEN, J. [1983]. "Register windows for SOAR," in *Smalltalk On A RISC: Architectural Investigations*, Proc. of CS 292R (April) 126–140. (p. 451)

BLOCH, E. [1959]. "The engineering design of the Stretch computer," *Proc. Fall Joint Computer Conf.*, 48–59. (p. 338)

BORRILL, P. L. [1986]. "32-bit buses—An objective comparison," *Proc. Buscon 1986 West*, San Jose, Calif., 138–145. (p. 533)

BOUKNIGHT, W. J, S. A. DENEBERG, D. E. MCINTYRE, J. M. RANDALL, A. H. SAMEH, AND D. L. SLOTNICK [1972]. "The Illiac IV system," *Proc. IEEE* 60:4, 369–379. Also appears in D. P. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples* (1982), 306–316. (p. 570)

BRADY, J. T. [1986]. "A theory of productivity in the creative process," *IEEE CG&A* (May) 25–34. (p. 560)

BRENT, R. P. AND H. T. KUNG [1982] "A regular layout for parallel adders," *IEEE Trans. on Computers* C-31, 260–264. (p. A-59)

BRODERSEN, R. W. [1989]. "Evolution of VLSI signal-processing circuits," *Proc. Decennial Caltech Conf. on VLSI* (March) 43–46, The MIT Press, Pasadena, Calif. (p. 590)

BUCHER, I. Y. [1983]. "The computational speed of supercomputers," *Proc. SIGMETRICS Conf. on Measuring and Modeling of Computer Systems,* ACM (August) 151–165. (p. 395)

BUCHER, I. Y. AND A. H. HAYES [1980]. "I/O Performance measurement on Cray-1 and CDC 7000 computers," *Proc. Computer Performance Evaluation Users Group, 16th Meeting,* NBS 500-65, 245–254. (p. 562)

BUCHOLTZ, W. [1962]. *Planning a Computer System: Project Stretch,* McGraw-Hill, New York. (p. 338)

BURKS, A. W., H. H. GOLDSTINE, AND J. VON NEUMANN [1946]. "Preliminary discussion of the logical design of an electronic computing instrument," Report to the U.S. Army Ordnance Department, p. 1; also appears in *Papers of John von Neumann,* W. Aspray and A. Burks, eds., The MIT Press, Cambridge, Mass. and Tomash Publishers, Los Angeles, Calif., 1987, 97–146. (p. 24)

CALLAHAN, D., J. DONGARRA, AND D. LEVINE [1988]. "Vectorizing compilers: A test suite and results," *Supercomputing '88,* ACM/IEEE (November), Orlando, Fla., 98–105. (p. 377)

CASE, R. P. AND A. PADEGS [1978]. "The architecture of the IBM System/370," *Comm. ACM* 21:1, 73–96. Also appears in D. P. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples* (1982), McGraw-Hill, New York, 830–855. (pp. 186, 485)

CENSIER, L. M. AND P. FEAUTRIER [1978]. "A new solution to the coherence problem in multicache systems," *IEEE Trans. on Computers* C-27:12 (December) 1112–1118. (p. 487)

CHAITIN, G. J., M. A. AUSLANDER, A. K. CHANDRA, J. COCKE, M. E. HOPKINS, AND P. W. MARKSTEIN [1982]. "Register allocation via coloring," *Computer Languages* 6, 47–57. (p. 130)

CHARLESWORTH, A. E. [1981]. "An approach to scientific array processing: The architecture design of the AP-120B/FPS-164 family," *Computer* 14:12 (December) 12–30. (p. 340)

CHEN, P. [1989]. *An Evaluation of Redundant Arrays of Inexpensive Disks Using an Amdahl 5890,* M. S. Thesis, Computer Science Division, Tech. Rep. UCB/CSD 89/506. (p. 507)

CHEN, S. [1983]. "Large-scale and high-speed multiprocessor system for scientific applications," *Proc. NATO Advanced Research Work on High Speed Computing* (June); also in K. Hwang, ed., "Supercomputers: Design and applications," *IEEE* (August) 1984. (p. 394)

CHEN, T. C. [1980]. "Overlap and parallel processing" in *Introduction to Computer Architecture,* H. Stone, ed., Science Research Associates, Chicago, 427–486. (p. 339)

CHOW, F. C. [1983]. *A Portable Machine-Independent Global Optimizer—Design and Measurements,* Ph. D. Thesis, Stanford Univ. (December). (p. 130)

CHOW, F. C. AND J. L. HENNESSY [1984]. "Register allocation by priority-based coloring," *Proc. SIGPLAN '84 Compiler Construction* (*ACM SIGPLAN Notices* 19:6, June) 222–232. (p. 130)

CHOW, F., M. HIMELSTEIN, E. KILLIAN, AND L. WEBER [1986]. "Engineering a RISC compiler system," *Proc. COMPCON* (March), San Francisco, 132–137. (p. 197)

CLARK, D. W. [1983]. "Cache performance of the VAX-11/780," *ACM Trans. on Computer Systems* 1:1, 24–37. (p. 486)

CLARK, D. W. [1987]. "Pipelining and performance in the VAX 8800 processor," *Proc. Second Conf. on Architectural Support for Programming Languages and Operating Systems*, IEEE/ACM (March), Palo Alto, Calif., 173–177. (p. 272)

CLARK, D. W. AND H. LEVY [1982]. "Measurement and analysis of instruction set use in the VAX-11/780," *Proc. Ninth Symposium on Computer Architecture* (April), Austin, Tex., 9–17. (p. 188)

CLARK, D. W. AND J. S. EMER [1985]. "Performance of the VAX-11/780 translation buffer: Simulation and measurement," *ACM Trans. on Computer Systems* 3:1, 31–62. (p. 486)

CLARK, D. W. AND W. D. STRECKER [1980]. "Comments on 'the case for the reduced instruction set computer', " *Computer Architecture News* 8:6 (October) 34–38. (p. 130)

CLARK, D. W., P. J. BANNON, AND J. B. KELLER [1988]. "Measuring VAX 8800 performance with a histogram hardware monitor," *Proc. 15th Annual Symposium on Computer Architecture* (May–June), Honolulu, Hawaii, 176–185. (pp. 213, 486)

COCKE, J. AND J. T. SCHWARTZ [1970]. *Programming Languages and Their Compilers*, Courant Institute, New York Univ., New York City. (p. 130)

COCKE, J., AND J. MARKSTEIN [1980]. "Measurement of code improvement algorithms," *Information Processing* 80, 221–228. (p. 130)

CODD, E. F. [1962]. "Multiprogramming," in F.L. Alt and M. Rubinoff, *Advances in Computers*, vol. 3, Academic Press, New York, 82. (p. 241)

CODY, W. J. [1988]. "Floating point standards: Theory and practice," in *Reliability in Computing: The Role of Interval Methods in Scientific Computing*, R. E. Moore, (ed.), Academic Press, Boston, Mass., 99–107. (p. A-12)

CODY, W. J., J. T. COONEN, D. M. GAY, K. HANSON, D. HOUGH, W. KAHAN, R. KARPINSKI, J. PALMER, F. N. RIS, AND D. STEVENSON [1984]. "A proposed radix- and word-length-independent standard for floating-point arithmetic," *IEEE Micro* 4:4, 86–100. (p. A-12)

COHEN, D. [1981]. "On holy wars and a plea for peace," *Computer* 14:10 (October) 48–54. (p. 95)

COLWELL, R. P, C. Y. HITCHCOCK, III, E. D. JENSEN, H. M. B. SPRUNT, AND C. P. KOLLAR, [1985]. "Computers, complexity, and controversy," *Computer* 18:9 (September) 8–19. (p. 125)

COLWELL, R. P., R. P. NIX, J. J. O'DONNELL, D. B. PAPWORTH, AND B. K. RODMAN [1987]. "A VLIW architecture for a trace scheduling compiler," *Proc. Second Conf. on Architectural Support for Programming Languages and Operating Systems*, IEEE/ACM (March), Palo Alto, Calif., 180–192. (p. 340)

CONTI, C., D. H. GIBSON, AND S. H. PITKOWSKY [1968]. "Structural aspects of the System/360 Model 85, part I: General organization," *IBM Systems J.* 7:1, 2–14. (pp. 77, 486)

COONEN, J. [1984]. *Contributions to a Proposed Standard for Binary Floating-Point Arithmetic*, Ph.D. Thesis, Univ. of Calif., Berkeley. (p. A-29)

CRAWFORD, J. H AND P. P. GELSINGER [1987]. *Programming the 80386*, Sybex, Alameda, Calif. (pp. 188, 446)

CURNOW, H. J. AND B. A. WICHMANN [1976]. "A synthetic benchmark," *The Computer J.* 19:1. (p.77)

DAVIDSON, E. S. [1971]. "The design and control of pipelined function generators," *Proc. Conf. on Systems, Networks, and Computers*, IEEE (January), Oaxtepec, Mexico, 19–21. (p. 339)

DAVIDSON, E. S., A. T. THOMAS, L. E. SHAR, AND J. H. PATEL [1975]. "Effective control for pipelined processors," *COMPCON, IEEE* (March), San Francisco, 181–184. (p. 339)

DEHNERT, J. C., P. Y.-T. HSU, AND J. P. BRATT [1989]. "Overlapped loop support on the Cydra 5," *Proc. Third Conf. on Architectural Support for Programming Languages and Operating Systems* (April), IEEE/ACM, Boston, 26–39. (p. 340)

DEROSA, J., R. GLACKEMEYER, AND T. KNIGHT [1985]. "Design and implementation of the VAX 8600 pipeline," *Computer* 18:5 (May) 38–48. (p. 328)

DEWITT, D. J., R. FINKEL, AND M. SOLOMON [1984]. "The CRYSTAL multicomputer: Design and implementation experience, Computer Sciences Tech. Rep. No. 553, University of Wisconsin-Madison, September. (p. 590)

DIGITAL EQUIPMENT CORPORATION [1987]. *Digital Technical J.* 4 (March), Hudson, Mass. (This entire issue is devoted to the VAX 8800 processor.) (p. 341)

DITZEL, D. R. [1981]. "Reflections on the high-level language Symbol computer system," *Computer* 14:7 (July) 55–66. (p. 129)

DITZEL, D. R. AND D. A. PATTERSON [1980]. "Retrospective on high-level language computer architecture," in *Proc. Seventh Annual Symposium on Computer Architecture,* La Baule, France (June) 97–104. (p. 130)

DITZEL, D. R. AND H. R. MCLELLAN [1987]. "Branch folding in the CRISP microprocessor: Reducing the branch delay to zero," *Proc. 14th Symposium on Computer Architecture* (June), Pittsburgh, 2–7. (p. 339)

DITZEL, D. R., AND H. R. MCLELLAN [1982]. "Register allocation for free: The C machine stack cache," *Symposium on Architectural Support for Programming Languages and Operating Systems* (March 1–3), Palo Alto, Calif., 48–56. (p. 487)

DOHERTY, W. J. AND R. P. KELISKY [1979]. "Managing VM/CMS systems for user effectiveness," *IBM Systems J.* 18:1, 143–166. (p. 560)

DONGARRA, J. J. [1986]. "A survey of high performance computers," *COMPCON, IEEE* (March) 8–11. (p. 394)

EARLE, J. G. [1965]. "Latched carry-save adder," *IBM Technical Disclosure Bull.* 7 (March) 909–910. (p. 254)

EGGERS, S. [1989]. *Simulation Analysis of Data Sharing in Shared Memory Multiprocessors* , Ph.D. Thesis, Univ. of California, Berkeley, Computer Science Division Tech. Rep. UCB/CSD 89/501 (April). (p. 487)

ELDER, J., A. GOTTLIEB, C. K. KRUSKAL, K. P. MCAULIFFE, L. RANDOLPH, M. SNIR, P. TELLER, AND J. WILSON [1985]. "Issues related to MIMD shared-memory computers: The NYU Ultracomputer approach," *Proc. 12th Int'l Symposium on Computer Architecture* (June), Boston, Mass., 126–135. (p. 589)

ELLIS, J. R., J. A. FISHER, J. C. RUTTENBERG, AND A. NICHOLAU [1984]. "Parallel processing: A smart compiler and a dumb machine," *Proc. SIGPLAN Conf. on Compiler Construction* (June), Montreal, Canada, 37–47. (p. 340)

ELSHOFF, J. L. [1976]. "An analysis of some commercial PL/I programs," *IEEE Trans. on Software Engineering* SE-2 2 (June) 113–120. (p. 130)

EMER, J. S. AND D. W CLARK [1984]. "A characterization of processor performance in the VAX-11/780," *Proc. 11th Symposium on Computer Architecture* (June), Ann Arbor, Mich., 301–310. (pp. 189, 213, 342, 486)

E•SUN MICROSYSTEMS [1989]. *The SPARC Architectural Manual,* Version 8, Part No. 800-1399-09, August 25, 1989.

FABRY, R. S. [1974]. "Capability based addressing," *Comm. ACM* 17:7 (July) 403–412. (p. 485)

FAZIO, D. [1987]. "It's really much more fun building a supercomputer than it is simply inventing one," *COMPCON, IEEE* (February) 102-105. (p. 394)

FEIERBACK, G AND D. STEVENSON [1979]. "The Illiac-IV," in *Infotech State of the Art Report on Supercomptuers,* Maidenhead, England. This data also appears in D. P. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples* (1982), McGraw-Hill, New York, 268–269. (p. 556)

FISHER, J. A. [1983]. "Very long instruction word architectures and ELI-512," *Proc. Temth Symposium on Computer Architecture* (June), Stockholm, Sweden. (p. 340)

FLEMMING, P. J. AND J. J. WALLACE [1986]. "How not to lie with statistics: The correct way to summarize benchmarks results," *Comm. ACM* 29:3 (March) 218–221. (p. 78)

FLYNN, M. J. [1966]. "Very high-speed computing systems," *Proc. IEEE* 54:12 (December) 1901–1909. (pp. 351, 591)

FOLEY, J. D. AND A. VAN DAM [1982]. *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, Reading, Mass. (p. 561)

FOSTER, C. C. AND E. M. RISEMAN [1972]. "Percolation of code to enhance parallel dispatching and execution," *IEEE Trans. on Computers* C-21:12 (December) 1411–1415. (p. 340)

FOSTER, C. C., R. H. GONTER, AND E. M. RISEMAN [1971]. "Measures of opcode utilization," *IEEE Trans. on Computers* 13:5 (May) 582–584. (p. 129)

FRANK, P. D. [1987]. "Advances in Head Technology," presentation at *Challenges in Winchester Technology* (December 15), Santa Clara Univ. (p. 561)

FRANK, S. J. [1984]. "Tightly coupled multiprocessor systems speed memory access times," *Electronics* 57:1 (January) 164–169. (p. 487)

FREIMAN, C. V. [1961]. "Statistical analysis of certain binary division algorithms," *Proc. IRE* 49:1, 91–103. (p. A-59)

FRIESENBORG, S. E. AND R. J. WICKS [1985]. "DASD expectations: The 3380, 3380-23, and MVS/XA," Tech. Bulletin GG22-9363-02 (July 10), Washington Systems Center. (p. 554)

FULLER, S. H. [1976]. "Price/performance comparison of C.mmp and the PDP-11," *Proc. Third Annual Symposium on Computer Architecture* (Texas, January 19–21), 197–202. (p. 80)

FULLER, S. H. AND W. E. BURR [1977]. "Measurement and evaluation of alternative computer architectures," *Computer* 10:10 (October) 24–35. (p. 78)

GAGLIARDI, U. O. [1973]. "Report of workshop 4–software-related advances in computer hardware," *Proc. Symposium on the High Cost of Software*, Menlo Park, Calif., 99–120. (p. 129)

GAJSKI, D., D. KUCK, D. LAWRIE, AND A. SAMEH [1983]. "CEDAR—A large scale multiprocessor," *Proc. Int'l Conf. on Parallel Processing* (August) 524–529. (p. 589)

GARNER, R., A. AGARWAL, F. BRIGGS, E. BROWN, D. HOUGH, B. JOY, S. KLEIMAN, S. MUNCHNIK, M. NAMJOO, D. PATTERSON, J. PENDLETON, AND R. TUCK [1988]. "Scaleable processor architecture (SPARC)," *COMPCON, IEEE* (March), San Francisco, 278–283. (p. 190)

GEHRINGER, E. F., D. P. SIEWIOREK, AND Z. SEGALL [1987]. *Parallel Processing: The Cm* Experience*, Digital Press, Bedford, Mass. (p. 587)

GIBSON, D. H. [1967]. "Considerations in block–oriented systems design," *AFIPS Conf. Proc.* 30, SJCC, 75–80. (p. 486)

GIBSON, J. C. [1970]. "The Gibson mix," Rep. TR. 00.2043, IBM Systems Development Division, Poughkeepsie, N.Y. (Research done in 1959.) (p. 77)

GOLDBERG, D. [1989]. "Floating-point and computer systems," *Xerox Tech. Rep.* CSL-89-9. A version of this paper will appear in *Computing Surveys.* (p. A-29)

GOLDBERG, I. B. [1967]. "27 bits are not enough for 8-digit accuracy," *Comm. ACM* 10:2, 105–106. (p. A-60)

GOLDSTEIN, S. [1987]. "Storage performance—an eight year outlook," Tech. Rep. TR 03.308-1 (October), Santa Teresa Laboratory, IBM, San Jose, Calif. (p. 561)

GOLDSTINE, H. H. [1972]. *The Computer: From Pascal to von Neumann,* Princeton University Press, Princeton, N.J. (p. 25)

GOODMAN, J. R. [1983]. "Using cache memory to reduce processor memory traffic," *Proc. Tenth Annual Symposium on Computer Architecture* (June 5–7), Stockholm, Sweden, 124–131. (p. 487)

GOODMAN, J. R. and M.-C. Chiang [1984]. "The use of static column RAM as a memory hierarchy," *Proc. 11th Annual Symposium on Computer Architecture* (June 5–7), Ann Arbor, Mich., 167–174. (p. 488)

GOSLING, J. B. [1980]. *Design of Arithmetic Units for Digital Computers,* Springer-Verlag NewYork, Inc., New York. (p. A-61)

GRAY, W. P. [1989]. Memorandum of Decision, No. C-84-20799-WPG, U.S. District Court for the Northern District of California (February 7, 1989). (p. 244)

GROSS, T. R. [1983]. *Code Optimization of Pipeline Constraints,* Ph.D. Thesis (December), Computer Systems Lab., Stanford Univ. (p. 342)

HALBERT, D. C. AND P. B. KESSLER [1980]. "Windows of overlapping register frames," *CS 292R Final Reports* (June) 82–100. (p. 489)

HAMACHER, V. C., Z. G. VRANESIC, AND S. G. ZAKY [1984]. *Computer Organization,* 2nd ed., McGraw-Hill, New York. (p. A-61)

HAUCK, E. A., AND B. A. DENT [1968]. "Burroughs' B6500/B7500 stack mechanism," *Proc. AFIPS SJCC,* 245–251. (p. 131)

HENLY, M. AND B. MCNUTT [1989]. "DASD I/O characteristics: A comparison of MVS to VM," Tech. Rep. TR 02.1550 (May), IBM, General Products Division, San Jose, Calif. (pp. 80, 562)

HENNESSY, J. [1984]. "VLSI processor architecture," *IEEE Trans. on Computers* C-33:11 (December) 1221–1246. (p. 190)

HENNESSY, J. [1985]. "VLSI RISC processors," *VLSI Systems Design* VI:10 (October) 22–32. (p. 191)

HENNESSY, J. L. AND T. R. GROSS [1983]. "Postpass code optimization of pipeline constraints," *ACM Trans. on Programming Languages and Systems* 5:3 (July) 422–448. (p. 342)

HENNESSY, J., N. JOUPPI, F. BASKETT, AND J. GILL [1981]. "MIPS: A VLSI processor architecture," *Proc. CMU Conf. on VLSI Systems and Computations* (October), Computer Science Press, Rockville, Md. (p. 191)

HENNESSY, J. L., N. JOUPPI, F. BASKETT, T. R. GROSS, AND J. GILL [1982]. "Hardware/software tradeoffs for increased performance," *Proc. Symposium on Architectural Support for Programming Languages and Operating Systems* (March), 2–11. (p. 131)

HENNESSY, J. [1984]. "VLSI processor architecture," *IEEE Trans. on Computers* C-33:11 (December) 1221–1246. (p. 189)

HILL, M. D. [1987]. *Aspects of Cache Memory and Instruction Buffer Performance,* Ph.D. Thesis, Univ. of California at Berkeley Computer Science Division, Tech. Rep. UCB/CSD 87/381 (November). (p. 489)

HILL, M. D. [1988]. "A case for direct mapped caches," *Computer* 21:12 (December) 25–40. (p. 489)

HILLIS, W. D. [1985]. *The Connection Machine,* The MIT Press, Cambridge, Mass. (p. 591)

HINTZ, R. G. AND D. P. TATE [1972]. "Control data STAR-100 processor design," *COMPCON, IEEE* (September) 1–4. (p. 396)

HOCKNEY, R. W. AND C. R. JESSHOPE [1988]. *Parallel Computers-2, Architectures, Programming and Algorithms,* Adam Hilger Ltd., Bristol, England and Philadelphia. (p. 591)

HOLLAND, J. H. [1959]. "A universal computer capable of executing an arbitrary number of subprograms simultaneously," *Proc. East Joint Computer Conf.* 16, 108–113. (p. 591)

HOLLINGSWORTH, W., H. SACHS AND A. J. SMITH [1989]. "The Clipper processor: Instruction set architecture and implementation," *Comm. ACM* 32:2 (February), 200–219. (p. 80)

HORD, R. M. [1982]. *The Illiac-IV, The First Supercomputer,* Computer Science Press, Rockville, Md. (p. 591)

HOWARD, J. H. ET AL. [1988]. "Scale and performance in a distributed file system," *ACM Trans. on Computer Systems* 6:1, 51–81. (p. 512)

HUGUET, M. AND T. LANG [1985]. "A reduced register file for RISC architectures," *Computer Architecture News* 13:4 (September) 22–31. (p. 489)

HWANG, K. [1979]. *Computer Arithmetic: Principles, Architecture, and Design,* Wiley, New York. (p. A-61)

HWU, W.-M. AND Y. PATT [1986]. "HPSm, a high performance restricted data flow architecture having minimum functionality," *Proc. 13th Symposium on Computer Architecture* (June), Tokyo, 297–307. (p. 339)

IBM [1982]. *The Economic Value of Rapid Response Time*, GE20-0752-0 White Plains, N.Y., 11–82. (p. 560)

IEEE [1985]. "IEEE standard for binary floating-point arithmetic," *SIGPLAN Notices* 22:2, 9–25. (p. A-12)

IMPRIMIS [1989]. "Imprimis Product Specification, 97209 Sabre Disk Drive IPI-2 Interface 1.2 GB," Document No. 64402302 (May). (p. 558)

[1989]. *i860 64-Bit Microprocessor Programmer's Reference Manual.* (E-24)

JORDAN, K. E. [1987]. "Performance comparison of large-scale scientific computers: Scalar mainframes, mainframes with vector facilities, and supercomputers," *Computer* 20:3 (March) 10–23. (p. 395)

JOUPPI N. P. AND D. W. WALL [1989]. "Available instruction-level parallelism for superscalar and superpipelined machines," *Proc. Third Conf. on Architectural Support for Programming Languages and Operating Systems,* IEEE/ACM (April), Boston, 272–282. (p. 340)

KAHAN, W. [1968]. "7094-II system support for numerical analysis," *SHARE Secretarial Distribution* SSD-159. (p. A-60)

KAHANER, D. K. [1988]. "Benchmarks for 'real' programs," *SIAM News* (November). (p. A-57)

KAHN, R. E. [1972]. "Resource-sharing computer communication networks," *Proc. IEEE* 60:11 (November) 1397–1407. (p. 561)

KANE, G. [1986]. *MIPS R2000 RISC Architecture,* Prentice Hall, Englewood Cliffs, N.J. (p. 190)

KANE, G. [1988]. *MIPS RISC Architecture,* Prentice-Hall, Englewood Cliffs, N. J. (E-24)

KATZ, R. H., D. A. PATTERSON, AND G. A. GIBSON [1990]. "Disk system architectures for high performance computing," *Proc. IEEE* 78:2 (February). (p. 561)

KATZ, R. H., S. EGGERS, D. A. WOOD, C. PERKINS, AND R. G. SHELDON [1985]. "Implementing a cache consistency protocol," *Proc. 12th Annual Symposium on Computer Architecture*, 276–283. (p. 487)

KELLER R. M. [1975]. "Look-ahead processors," *ACM Computing Surveys* 7:4 (December) 177–195. (p. 339)

KELLY, E. [1988]. "'SCRAM Cache' in Sun-4/110 beats traditional caches," *Sun Technology* 1:3 (Summer) 19–21. (p. 487)

KILBURN, T., D. B. G. EDWARDS, M. J. LANIGAN, F. H. SUMNER [1962]. "One-level storage system," *IRE Transactions on Electronic Computers* EC-11 (April) 223–235. Also appears in D. P. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples* (1982), McGraw-Hill, New York, 135–148. (pp. 26, 487)

KIM, M. Y. [1986]. "Synchronized disk interleaving," *IEEE Trans. on Computers* C-35:11 (November). (p. 561)

KNUTH, D. [1981]. *The Art of Computer Programming,* vol II, 2nd ed., Addison-Wesley, Reading, Mass. (p. A-61)

KNUTH, D. E. [1971]. "An empirical study of FORTRAN programs," *Software Practice and Experience,* Vol. 1, 105–133. (p. 27)

KOGGE, P. M. [1981]. *The Architecture of Pipelined Computers,* McGraw-Hill, New York. (pp. 339, A-44)

KOHN, L. AND S.-W. FU, [1989]. "A 1,000,000 transistor microprocessor," *IEEE Int'l Solid-State Circuits Conf.,* 54–55. (p. A-19)

KROFT, D. [1981]. "Lockup-free instruction fetch/prefetch cache organization," *Proc. Eighth Annual Symposium on Computer Architecture* (May 12–14), Minneapolis, Minn., 81–87. (p. 487)

KUCK, D., P. P. BUDNIK, S.-C. CHEN, D. H. LAWRIE, R. A. TOWLE, R. E. STREBENDT, E. W. DAVIS, JR., J. HAN, P. W. KRASKA, Y. MURAOKA [1974]. "Measurements of parallelism in ordinary FORTRAN programs," *Computer* 7:1 (January) 37–46. (p. 395)

KUHN, R. H. AND D. A. PADUA, EDS. [1981]. *Tutorial on Parallel Processing,* IEEE. (p. 590)

KUNG, H. T. [1982]. "Why systolic architectures?," *IEEE Computer* 15:1, 37–46. (p. 590)

KUNKEL, S. R. AND J. E. SMITH [1986]. "Optimal pipelining in supercomputers," *Proc. 13th Symposium on Computer Architecture* (June), Tokyo, 404–414. (p. 339)

LAM, M. [1988]. "Software pipelining: An effective scheduling technique for VLIW machines," *SIGPLAN Conf. on Programming Language Design and Implementation,* ACM (June), Atlanta, Ga., 318–328. (p. 340)

LAMPSON, B. W. [1982]. "Fast procedure calls," *Symposium on Architectural Support for Programming Languages and Operating Systems* (March 1–3), Palo Alto, Calif., 66–75. (p. 487)

LARSON, JUDGE E. R. [1973]. "Findings of Fact, Conclusions of Law, and Order for Judgment," File No. 4–67, Civ. 138, *Honeywell v. Sperry Rand and Illinois Scientific Development,* U.S. District Court for the District of Minnesota, Fourth Division (October 19). (p. 24)

LEE, R. [1989]. "Precision architecture," *Computer* 22:1 (January) 78–91. (p. 190)

LEINER, A. L. [1954]. "System specifications for the DYSEAC," *J. ACM* 1:2 (April) 57–81. (p. 561)

LEINER, A. L. AND S. N. ALEXANDER [1954]. "System organization of the DYSEAC," *IRE Trans. of Electronic Computers* EC-3:1 (March) 1–10. (p. 561)

LEVY, H. M. AND R. H. ECKHOUSE, JR. [1989]. *Computer Programming and Architecture: The VAX,* 2nd ed., Digital Press, Bedford, Mass. 358–372. (pp. 188, 243)

LEVY, J. V. [1978]. "Buses: The skeleton of computer structures," in *Computer Engineering: A DEC View of Hardware Systems Design,* C. G. Bell, J. C. Mudge, and J. E. McNamara, eds., Digital Press, Bedford, Mass. (p. 561)

LINCOLN, N. R. [1982]. "Technology and design tradeoffs in the creation of a modern supercomputer," *IEEE Trans. on Computers* C-31:5 (May) 363–376. (p. 393)

LIPOVSKI, A. G. AND A. TRIPATHI [1977]. "A reconfigurable varistructure array processor," *Proc. 1977 Int'l Conf. of Parallel Processing* (August), 165–174. (p. 590)

LIPTAY, J. S. [1968]. "Structural aspects of the System/360 Model 85, part II: The cache," *IBM Systems J.* 7:1, 15–21. (p. 486)

LOVETT, T. AND S. THAKKAR [1988]. "The Symmetry multiprocessor system," *Proc. 1988 Int'l Conf. of Parallel Processing,* University Park, Pennsylvania, 303–310. (p. 589)

LUBECK, O., J. MOORE, AND R. MENDEZ [1985]. "A benchmark comparison of three supercomputers: Fujitsu VP-200, Hitachi S810/20, and CRAY X-MP/2," *Computer* 18:12 (December) 10–24. (pp. 75, 395)

LUNDE, A. [1977]. "Empirical evaluation of some features of instruction set processor architecture," *Comm. ACM* 20:3 (March) 143–152. (p. 129)

MABERLY, N. C. [1966]. *Mastering Speed Reading,* New American Library, Inc., New York. (p. 513)

MAGENHEIMER, D. J., L. PETERS, K. W. PETTIS AND D. ZURAS [1988]. "Integer multiplication and division on the HP Precision Architecture," *IEEE Trans. on Computers,* 37:8, 980–990. (p. E-9)

MAGENHEIMER, D. J., L. PETERS, K. W. PETTIS, AND D. ZURAS, [1988]. "Integer multiplication and division on the HP Precision Architecture," *IEEE Trans. on Computers* 37:8, 980–990. (p. A-11)

MCCALL, K. [1983]. "The Smalltalk-80 benchmarks," *Smalltalk 80: Bits of History, Words of Advice,* G. Krasner, ed., Addison-Wesley, Reading, Mass., 153–174. (p. 451)

MCCREIGHT, E. [1984]. "The Dragon computer system: An early overview," Tech. Rep. Xerox Corp. (September). (p. 487)

MCFARLING, S. [1989]. "Program optimization for instruction caches," *Proc. Third Int'l Conf. on Architectural Support for Programming Languages and Operating Systems* (April 3–6), Boston, Mass., 183–191. (p. 496)

MCFARLING, S. AND J. HENNESSY [1986]. "Reducing the cost of branches," *Proc. 13th Symposium on Computer Architecture* (June), Tokyo, 396–403. (p. 340)

MCKEEMAN, W. M. [1967]. "Language directed computer design," *Proc. 1967 Fall Joint Computer Conf.,* Washington, D.C., 413–417. (p. 128)

MCKEVITT, J., ET AL. [1977]. *8086 Design Report*, internal memorandum. (p. 229)

MCMAHON, F. M. [1986]. "The Livermore FORTRAN kernels: A computer test of numerical performance range," Tech. Rep. UCRL-55745, Lawrence Livermore National Laboratory, Univ. of California, Livermore, Calif. (December). (p. 78)

MEAD, C. AND L. CONWAY [1980]. *Introduction to VLSI Systems,* Addison-Wesley, Reading, Mass. (p. A-59)

MENABREA, L. F. [1842]. "Sketch of the analytical engine invented by Charles Babbage," Bibiothèque Universelle de Genève (October). (p. 589)

METCALFE, R. M. AND D. R. BOGGS [1976]. "Ethernet: Distributed packet switching for local computer networks," *Comm. ACM* 19:7 (July) 395–404. (p. 560)

MEYERS, G. J. [1978]. "The evaluation of expressions in a storage-to-storage architecture," *Computer Architecture News* 7:3 (October), 20–23. (p. 127)

MEYERS, G. J. [1982]. *Advances in Computer Architecture*, 2nd ed., Wiley, N.Y. (p. 129)

MIRANKER, G. S., J. RUBENSTEIN, AND J. SANGUINETTI [1988]. "Squeezing a Cray-class supercomputer into a single-user package," *COMPCON, IEEE* (March) 452–456. (p. 395)

MITCHELL, D. [1989]. "The Transputer: The time is now," *Computer Design*, RISC supplement, 40–41 (November). (p. 570)

MIURA, K. AND K. UCHIDA [1983]. "FACOM vector processing system: VP100/200," *Proc. NATO Advanced Research Work on High Speed Computing* (June); also in K. Hwang, ed., "Supercomputers: Design and applications," *IEEE* (August 1984) 59–73. (p. 394)

MOORE, B., A. PADEGS, R. SMITH, AND W. BUCHOLZ [1987]. "Concepts of the System/370 vector architecture," *Proc. 14th Symposium on Computer Architecture* (June), ACM/IEEE, Pittsburgh, Pa., 282–292. (p. 394)

MORSE, S., B. RAVENAL, S. MAZOR, AND W. POHLMAN [1980]. "Intel Microprocessors—8008 to 8086," *Computer* 13:10 (October). (p. 188)

MOTOROLA [1988]. *MC88100 RISC Microprocessor User's Manual.* (E-19)

MOUSSOURIS, J., L. CRUDELE, D. FREITAS, C. HANSEN, E. HUDSON, S. PRZYBYLSKI, T. RIORDAN, AND C. ROWEN [1986]. "A CMOS RISC processor with integrated system functions," *Proc. COMPCON, IEEE* (March), San Francisco. (p. 189)

MUCHNICK, S. S. [1988]. "Optimizing compilers for SPARC," *Sun Technology* (Summer) 1:3, 64–77. (p. E-9)

NEWMAN, W. N. AND R. F. SPROULL [1979]. *Principles of Interactive Computer Graphics*, 2nd ed., McGraw-Hill, New York. (p. 561)

NGAI, T-F. AND M. J. IRWIN [1985]. "Regular, area-time efficient carry-lookahead adders," *Proc. Seventh IEEE Symposium on Computer Arithmetic,* 9–15. (p. A-59)

NICHOLAU, A. AND J. A. FISHER [1984]. "Measuring the parallelism available for very long instruction word architectures," *IEEE Trans. on Computers* C-33:11 (November) 968–976. (p. 340)

OUSTERHOUT, J. K. ET AL. [1985]. "A trace-driven analysis of the UNIX 4.2 BSD file system," *Proc. Tenth ACM Symposium on Operating Systems Principles*, Orcas Island, Wash., 15–24. (p. 538)

PADUA, D. AND M. WOLFE [1986]. "Advanced compiler optimizations for supercomputers," *Comm. ACM* 29:12 (December) 1184–1201. (p. 395)

PAPAMARCOS, M. AND J. PATEL [1984]. "A low coherence solution for multiprocessors with private cache memories," *Proc. of the 11th Annual Symposium on Computer Architecture* (June), Ann Arbor, Mich., 348–354. (p. 487)

PATTERSON, D. A. [1983]. "Microprogramming," *Scientific American* 248:3 (March), 36–43. (p. 244)

PATTERSON, D. A. [1985]. "Reduced Instruction Set Computers," *Comm. ACM* 28:1 (January) 8–21. (p. 189)

PATTERSON, D. A. AND C. H. SEQUIN [1981]. "Lockup-free instruction fetch/prefetch cache organization," *Proc. Eighth Annual Symposium on Computer Architecture* (May 12–14), Minneapolis, Minn., 443–458. (p. 487)

PATTERSON, D. A. AND D. R. DITZEL [1980]. "The case for the reduced instruction set computer," *Computer Architecture News* 8:6 (October), 25–33. (pp. 130, 189)

PATTERSON, D. A., G. A. GIBSON, AND R. H. KATZ [1987]. "A case for redundant arrays of inexpensive disks (RAID)," Tech. Rep. UCB/CSD 87/391, Univ. of Calif. Also appeared in *ACM SIGMOD Conf. Proc.*, Chicago, Illinois, June 1–3, 1988, 109–116. (p. 561)

PENG, V., S. SAMUDRALA, AND M. GAVRIELOV [1987]. "On the implementation of shifters, multipliers, and dividers in VLSI floating point units," *Proc. Eighth IEEE Symposium on Computer Arithmetic,* 95–102. (p. A-62)

PFISTER, G. F., W. C. BRANTLEY, D. A. GEORGE, S. L. HARVEY, W. J. KLEINFEKDER, K. P. MCAULIFFE, E. A. MELTON, V. A. NORTON, AND J. WEISS [1985]. "The IBM research parallel processor prototype (RP3): Introduction and architecture," *Proc. 12th Int'l Symposium on Computer Architecture* (June), Boston, Mass., 764–771. (p. 589)

PHISTER, M., JR. [1979]. *Data Processing Technology and Economics,* 2nd ed., Digital Press and Santa Monica Publishing Company. (p. 80)

PRZYBYLSKI, S. A. [1990]. *Cache Design: A Performance-Directed Approach,* Morgan Kaufmann Publishers, San Mateo, Calif. (p. 487)

PRZYBYLSKI, S. A., M. HOROWITZ, AND J. L. HENNESSY [1988]. "Performance tradeoffs in cache design," *Proc. 15th Annual Symposium on Computer Architecture* (May–June), Honolulu, Hawaii, 290–298. (p. 481)

RADIN, G. [1982]. "The 801 minicomputer," *Proc. Symposium Architectural Support for Programming Languages and Operating Systems* (March), Palo Alto, Calif. 39–47. (p. 189)

RAMAMOORTHY, C. V. AND H. F. LI [1977]. "Pipeline architecture," *ACM Computing Surveys* 9:1 (March) 61–102. (p. 339)

REDMOND, K. C. AND T. M. SMITH [1980]. *Project Whirlwind—The History of a Pioneer Computer,* Digital Press, Boston, Mass. (p. 25)

REIGEL, E. W., U. FABER, AND D. A. FISCHER, [1972]. "The Interpreter—a microprogrammable building block system," *Proc. AFIPS 1972 Spring Joint Computer Conf.* 40, 705–723. (p. 244)

ROBERTS, D., G. TAYLOR, AND T. LAYMAN [1990]. "An ECL RISC microprocessor designed for two-level cache," *IEEE COMPCON* (February). (p. 487)

ROBINSON, B. AND L. BLOUNT [1986]. "The VM/HPO 3880-23 performance results," IBM Tech. Bulletin, GG66-0247-00 (April), Washington Systems Center, Gathersburg, Md. (p. 553)

ROWEN, C., M. JOHNSON, and P. RIES [1988]. "The MIPS R3010 floating-point coprocessor," *IEEE Micro* 53–62 (June). (p. A-53)

RUSSELL, R. M. [1978]. "The CRAY-1 computer system," *Comm. ACM* 21:1 (January) 63–72. (pp. 393, 590)

RYMARCZYK, J. [1982]. "Coding guidelines for pipelined processors," *Proc. Symposium on Architectural Support for Programming Languages and Operating Systems,* IEEE/ACM (March), Palo Alto, Calif., 12–19. (p. 339)

SALEM, K. AND H. GARCIA-MOLINA [1986]. "Disk striping," *IEEE 1986 Int'l Conf. on Data Engineering*. (p. 561)

SAMPLES, A. D. AND P. N. HILFINGER [1988]. "Code reorganization for instruction caches," Tech. Rep. UCB/CSD 88/447 (October), Univ. of Calif., Berkeley. (p. 496)

SANTORO, M. R., G. BEWICK, and M. A. HOROWITZ [1989]. "Rounding algorithms for IEEE multipliers," *Proc. Ninth IEEE Symposium on Computer Arithmetic,* 176–183. (p. A-21)

SCHNECK, P. B. [1987]. *Supercomputer Architecture,* Kluwer Academic Publishers, Norwell, Mass. (p. 394)

SCOTT, N. R. [1985]. *Computer Number Systems and Arithmetic,* Prentice-Hall, Englewood Cliffs, N.J. (p. A-1)

SCRANTON, R. A., D. A. THOMPSON, AND D. W. HUNTER [1983]. "The access time myth," Tech. Rep. RC 10197 (45223) (September 21), IBM, Yorktown Heights, N.Y. (p. 561)

SEITZ, C. [1985]. "The Cosmic Cube," *Comm. ACM* 28:1 (January) 22–31. (p. 590)

SHURKIN, J. [1984]. *Engines of the Mind: A History of the Computer*, W. W. Norton, New York. (p. 25)

SHUSTEK, L. J. [1978]. "Analysis and performance of computer instruction sets," Ph.D. Thesis (May), Stanford Univ., Stanford, Calif. (p. 187)

SITES, R. [1979]. *Instruction Ordering for the CRAY-1 Computer,* Tech. Rep. 78-CS-023 (July), Dept. of Computer Science, Univ. of Calif., San Diego. (p. 339)

SITES, R. L., [1979]. "How to use 1000 registers," *Caltech Conf. on VLSI* (January). (p. 487)

SLATER, R. [1987]. *Portraits in Silicon,* The MIT Press, Cambridge, Mass. (p. 25)

SLOTNICK, D. L., W. C. BORCK, AND R. C. MCREYNOLDS [1962]. "The Solomon computer," *Proc. Fall Joint Computer Conf.* (December), Philadelphia, 97–107. (p. 589)

SMITH, A. AND J. LEE [1984]. "Branch prediction strategies and branch target buffer design," *Computer* 17:1 (January) 6–22. (p. 339)

SMITH, A. J. [1982]. "Cache memories," *Computing Surveys* 14:3 (September) 473–530. (p. 486)

SMITH, A. J. [1985]. "Disk cache—miss ratio analysis and design considerations," *ACM Trans. on Computer Systems* 3:3 (August) 161–203. (p. 538)

SMITH, A. J. [1986]. "Bibliography and readings on CPU cache memories and related topics," *Computer Architecture News* (January) 22–42. (p. 486)

SMITH, B. J. [1981]. "Architecture and applications of the HEP multiprocessor system," *Real-Time Signal Processing IV* 298 (August) 241–248. (p. 395)

SMITH, J. E. [1981]. "A study of branch prediction strategies," *Proc. Eighth Symposium on Computer Architecture* (May), Minneapolis, 135–148. (p. 339)

SMITH, J. E. [1984]. "Decoupled access/execute computer architectures," *ACM Trans. on Computer Systems* 2:4 (November), 289–308. (p. 340)

SMITH, J. E. [1988]. "Characterizing computer performance with a single number," *Comm. ACM* 31:10 (October) 1202–1206. (p. 78)

SMITH, J. E. [1989]. "Dynamic instruction scheduling and the Astronautics ZS-1," *Computer* 22:7 (July) 21–35. (p. 340)

SMITH, J. E. AND A. R. PLEZKUN [1988]. "Implementing precise interrupts in pipelined processors," *IEEE Trans. on Computers* 37:5 (May) 562–573. (p. 339)

SMITH, J. E. AND J. R. GOODMAN [1983]. "A study of instruction cache organizations and replacement policies," *Proc. Tenth Annual Symposium on Computer Architecture* (June 5–7), Stockholm, Sweden,, 132–137. (p. 490)

SMITH, J. E., G. E. DERMER, B. D. VANDERWARN, S. D. KLINGER, C. M. ROZEWSKI, D. L. FOWLER, K. R. SCIDMORE, J. P. LAUDON [1987]. "The ZS-1 central processor," *Proc. Second Conf. on Architectural Support for Programming Languages and Operating Systems,* IEEE/ACM (March), Palo Alto, Calif., 199–204. (p. 340)

SMITH, M. D., M. JOHNSON, AND M. A. HOROWITZ [1989]. "Limits on multiple instruction issue," *Proc. Third Conf. on Architectural Support for Programming Languages and Operating Systems,* IEEE/ACM (April), Boston, Mass., 290–302. (p. 341)

SMITH, W. R., R. R. RICE, G. D. CHESLEY, T. A. LALIOTIS, S. F. LUNDSTROM, M. A. CHALHOUN, L. D. GEROULD, AND T. C. COOK [1971]. "SYMBOL: A large experimental system exploring major hardware replacement of software," *Proc. AFIPS Spring Joint Computer Conf.*, 601–616. (p. 129)

SMOTHERMAN , M. [1989]. "A sequencing-based taxonomy of I/O systems and review of historical machines," *Computer Architecture News* 17:5 (September) 5–15. (pp. 241, 561)

SOHI, G. S., AND S. VAJAPEYAM [1989]. "Tradeoffs in instruction format design for horizontal architectures," *Proc. Third Conf. on Architectural Support for Programming Languages and Operating Systems,* IEEE/ACM (April), Boston, Mass. 15–25. (p. 341)

SPEC [1989]. "SPEC Benchmark Suite Release 1.0," October 2, 1989. (p. 48)

SPORER, M., F. H. MOSS AND C. J. MATHAIS [1988]. "An introduction to the architecture of the Stellar Graphics supercomputer," *COMPCON, IEEE* (March) 464–467. (p. 395)

STERN, N. [1980]. "Who invented the first electronic digital computer," *Annals of the History of Computing* 2:4 (October) 375–376. (p. 24)

STRAPPER, C. H. [1989]. "Fact and fiction in yield modelling," Special Issue of the *Microelectronics Journal* entitled *Microelectronics into the Nineties*, Oxford, UK; Elsevier (May). (p. 80)

STRAPPER, C. H., F. H. ARMSTRONG, AND K. SAJI [1983]. "Integrated circuit yield statistics," *Proc. IEEE* 71:4 (April) 453–470. (p. 80)

STRECKER, W. D. [1976]. "Cache memories for the PDP-11?," *Proc. Third Annual Symposium on Computer Architecture* (January), Pittsburgh, Penn., 155–158. (pp. 187, 486)

STRECKER, W. D. [1978]. "VAX-11/780: A virtual address extension to the PDP-11 family," *Proc. AFIPS National Computer Conf.* 47, 967-980. (128, 187)

STRECKER, W. D. AND C. G. BELL [1976]. "Computer structures: What have we learned from the PDP-11?," *Proc. Third Symposium on Computer Architecture*. (p. 187)

SUTHERLAND, I. E. [1963]. "Sketchpad: A man-machine graphical communication system," *Spring Joint Computer Conf.* 329. (p. 561)

SWAN, R. J., A. BECHTOLSHEIM, K. W. LAI, AND J. K. OUSTERHOUT [1977]. "The implementation of the Cm* multi-microprocessor," *Proc. AFIPS National Computing Conf.*, 645–654. (p. 589)

SWAN, R. J., S. H. FULLER, AND D. P. SIEWIOREK [1977]. "Cm*—A modular, multimicroprocessor," *Proc. AFIPS National Computer Conf.* 46, 637–644. (p. 590)

SWARTZ, J. T. [1980]. "Ultracomputers," *ACM Transactions on Programming Languages and Systems* 4:2, 484–521 (p. 592)

SWARTZLANDER, E., ED. [1980]. *Computer Arithmetic,* Dowden, Hutchison and Ross (distributed by Van Nostrand, New York). (p. A-59)

TAKAGI, N., H. YASUURA, AND S. YAJIMA [1985]."High-speed VLSI multiplication algorithm with a redundant binary addition tree," *IEEE Trans. on Computers* C-34:9, 789–796. (p. A-59)

TANENBAUM, A. S. [1978]. "Implications of structured programming for machine architecture," *Comm. ACM* 21:3 (March) 237–246. (p. 128)

TANG, C. K. [1976]. "Cache system design in the tightly coupled multiprocessor system," *Proc. 1976 AFIPS National Computer Conf.*, 749–753. (p. 487)

TAYLOR, G. S. [1981]. "Compatible hardware for division and square root," *Proc. Fifth IEEE Symposium on Computer Arithmetic,* 127–134. (p. A-62)

TAYLOR, G. S. [1985]. "Radix 16 SRT dividers with overlapped quotient selection stages," *Proc. Seventh IEEE Symposium on Computer Arithmetic,* 64–71. (p. A-56)

TAYLOR, G. S., P. N. HILFINGER, J. R. LARUS, D. A. PATTERSON, AND B. G. ZORN [1986]. "Evaluation of the SPUR Lisp architecture," *Proc. 13th Annual Symposium on Computer Architecture* (June 2–5), Tokyo, Japan, 444–452. (pp. 189, 451)

TAYLOR, G., P. HILFINGER, J. LARUS, D. PATTERSON, AND B. ZORN [1986]. "Evaluation of the SPUR LISP architecture," *Proc. 13th Symposium on Computer Architecture (*June), Tokyo. (p. E-15)

THACKER, C. P. AND L. C. STEWART [1987]. "Firefly: a multiprocessor workstation," *Proc. Second Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, Calif., 164–172. (p. 487)

THACKER, C. P., E. M. MCCREIGHT, B. W. LAMPSON, R. F. SPROULL, AND D. R. BOGGS [1982]. "Alto: A personal computer," in *Computer Structures: Principles and Examples*, D. P. Siewiorek, C. G. Bell, and A. Newell, eds., McGraw-Hill, New York, 549–572. (p. 560)

THADHANI, A. J. [1981]. "Interactive user productivity," *IBM Systems J.* 20:4, 407–423. (p. 560)

THISQUEN, J. [1988]. "Seek time measurements," *Amdahl Peripheral Products Division Tech. Rep.* (May). (p. 558)

THORLIN, J. F. [1967]. "Code generation for PIE (parallel instruction execution) computers," *Spring Joint Computer Conf.* (April), Atlantic City, N.J. (p. 339)

THORNTON, J. E. [1964]. "Parallel operation in Control Data 6600," *Proc. AFIPS Fall Joint Computer Conf.* 26, part 2, 33–40. (pp. 128, 339)

THORTON, J. E. [1970]. *Design of a Computer, the Control Data 6600,* Scott, Foresman, Glenview, Ill. (p. 339)

TJADEN, G. S. AND M. J. FLYNN [1970]. "Detection and parallel execution of independent instructions," *IEEE Trans. on Computers* C-19:10 (October) 889–895. (p. 340)

TOMASULO, R. M. [1967]. "An efficient algorithm for exploring multiple arithmetic units," *IBM J. of Research and Development* 11:1 (January) 25–33. (p. 339)

TRELEAVEN, P. C., D. R. BROWNBRIDGE, and R. P. HOPKINS [1982]. "Data-driven and demand-driven computer architectures," *Computing Surveys*, 14:1 (March) 93–143. (p. 590)

TROIANI, M., S. S. CHING, N. N. QUAYNOR, J. E. BLOEM, AND F. C. COLON OSORIO [1985]. "The VAX 8600 I Box, a pipelined implementation of the VAX architecture," *Digital Technical J.* 1 (August) 4–19. (p. 328)

TUCKER, S. G. [1967]. "Microprogram control for the System/360," *IBM Systems Journal* 6:4, 222–241. (p. 242)

UNGAR, D. M. [1987]. *The Design of a High Performance Smalltalk System*, The MIT Press Distinguished Dissertation Series, Cambridge, Mass. (p. 451)

UNGAR, D., R. BLAU, P. FOLEY, D. SAMPLES, AND D. PATTERSON [1984]. "Architecture of SOAR: Smalltalk on a RISC," *Proc. 11th Symposium on Computer Architecture* (June), Ann Arbor, Mich., 188–197. (p. 189)

UNGAR, D., R. BLAU, P. FOLEY, D. SAMPLES, AND D. PATTERSON [1984]. "Architecture of SOAR: Smalltalk on a RISC," *Proc. 11th Symposium on Computer Architecture* (June), Ann Arbor, Mich., 188–197. (p. E-15)

UNGER, S. H. [1958]. "A computer oriented towards spatial problems," *Proc. Institute of Radio Engineers* 46:10 (October) 1744–1750. (p. 589)

VON NEUMANN, J. [1945]. "First draft of a report on the EDVAC." Reprinted in W. Aspray and A. Burks, eds., *Papers of John von Neumann on Computing and Computer Theory* (1987), 17–82, The MIT Press, Cambridge, Mass. (p. 592)

WAKERLY, J. [1989]. *Microcomputer Architecture and Programming,* J. Wiley, New York. (p. 188)

WANG, E.-H., J.-L. BAER, AND H. M. LEVY [1989]. "Organization and performance of a two-level virtual-real cache hierarchy," *Proc. 16th Annual Symposium on Computer Architecture* (May 28–June 1), Jerusalem, Israel, 140–148. (p. 487)

WATANABE, T. [1987]. "Architecture and performance of the NEC supercomputer SX system," *Parallel Computing* 5, 247–255. (p. 394)

WATERS, F., ED. [1986]. *IBM RT Personal Computer Technology,* IBM, Austin, Tex., SA 23-1057. (p. 190)

WATSON, W. J. [1972]. "The TI ASC–A highly modular and flexible super computer architecture," *Proc. AFIPS Fall Joint Computer Conf.*, 221–228. (p. 393)

WEICKER, R. P. [1984]. "Dhrystone: A synthetic systems programming benchmark," *Comm. ACM* 27:10 (October) 1013–1030. (p. 47)

WEISS, S. AND J. E. SMITH [1984]. "Instruction issue logic for pipelined supercomputers," *Proc. 11th Symposium on Computer Architecture* (June), Ann Arbor, Mich., 110–118. (p. 339)

WEISS, S. AND J. E. SMITH [1987]. "A study of scalar compilation techniques for pipelined supercomputers," *Proc. Second Conf. on Architectural Support for Programming Languages and Operating Systems* (March), IEEE/ACM, Palo Alto, Calif., 105–109. (p. 340)

WESTE, N. AND K. ESHRAGHIAN [1985]. *Principles of CMOS VLSI Design,* Addison-Wesley, Reading, Mass. (p. A-59)

WHITBY-STREVENS C. [1985]. "The transputer," *Proc. 12th Int'l Symposium on Computer Architecture*, Boston, Mass. (June) 292–300. (p. 589)

WICHMANN, B. A. [1973]. *Algol 60 Compilation and Assessment*, Academic Press, New York. (p. 46)

WIECEK, C. [1982]. "A case study of the VAX 11 instruction set usage for compiler execution," *Proc. Symposium on Architectural Support for Programming Languages and Operating Systems* (March), IEEE/ACM, Palo Alto, Calif., 177–184. (p. 188)

WILKES, M. [1965]. "Slave memories and dynamic storage allocation," *IEEE Trans. Electronic Computers* EC-14:2 (April) 270–271. (p. 486)

WILKES, M. V. [1953]. "The best way to design an automatic calculating machine," in *Manchester University Computer Inaugural Conf.*, 1951, Ferranti, Ltd., London. (Not published until 1953.) Reprinted in "The Genesis of Microprogramming" in *Annals of the History of Computing* 8:116. (p. 241)

WILKES, M. V. [1982]. "Hardware support for memory protection: Capability implementations," *Proc. Symposium on Architectural Support for Programming Languages and Operating Systems* (March 1–3), Palo Alto, Calif., 107–116. (pp. 107, 486)

WILKES, M. V. [1985]. *Memoirs of a Computer Pioneer*, The MIT Press, Cambridge, Mass. (pp. 25, 241)

WILKES, M. V. AND J. B. STRINGER [1953]. "Microprogramming and the design of the control circuits in an electronic digital computer," *Proc. Cambridge Philosophical Society* 49:230–238. Also reprinted in D. P. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples* (1982), McGraw-Hill, New York, 158–163, and in "The Genesis of Microprogramming" in *Annals of the History of Computing* 8:116. (p. 248)

WILKES, M. V. AND W. RENWICK [1949]. *Report of a Conf. on High Speed Automatic Calculating Machines,* Cambridge, England. (p. 88)

WILKES, M. V., D. J. WHEELER, AND S. GILL [1951]. *The Preparation of Programs for an Electronic Digital Computer*, Addison-Wesley Press, Cambridge, Mass. (p. 24)

WILLIAMS, T. E., M. HOROWITZ, R. L. ALVERSON, AND T. S. YANG [1987]. "A self-timed chip for division," *Advanced Research in VLSI, Proc. 1987 Stanford Conf.,* The MIT Press, Cambridge, Mass. (p. A-46)

WILSON, A. W., JR. [1987]. "Hierarchical cache/bus architecture for shared memory multiprocessors," *Proc. 14th Int'l Symposium on Computer Architecture* (June), Pittsburg, Penn., 244–252. (p. 589)

WULF, W. [1981]. "Compilers and computer architecture," *Computer* 14:7 (July) 41–47. (p. 130)

WULF, W. A., R. LEVIN AND S. P. HAREISON [1981]. *Hydra/C.mmp: An Experimental Computer System,* McGraw-Hill, New York. (p. 485)

WULF, W. AND C. G. BELL [1972]. "C.mmp—A multi-mini-processor," *Proc. AFIPS Fall Joint Computing Conf.* 41, part 2, 765–777. (p. 590)

WULF, W. AND S. P. HAREISON [1978]. "Reflections in a pool of processors—An experience report on C.mmp/Hydra," *Proc. AFIPS 1978 National Computing Conf.* 48 (June), Anaheim, Calif. 939–951. (p. 589)

Index

Bold page numbers indicate term definitions.

DLX Standard Instruction Set

Instruction type / opcode	Instruction meaning
Data transfers	**Move data between registers and memory, or between the integer and FP or special registers; only memory address mode is 16-bit displacement + contents of an integer register**
LB, LBU, SB	Load byte, load byte unsigned, store byte
LH, LHU, SH	Load halfword, load halfword unsigned, store halfword
LW, SW	Load word, store word (to/from integer registers)
LF, LD, SF, SD	Load SP float, load DP float, store SP float, store DP float
MOVI2S, MOVS2I	Move from/to integer register to/from a special register
MOVF, MOVD	Copy one floating-point register or a DP pair to another register or pair
MOVFP2I, MOVI2FP	Move 32 bits from/to FP registers to/from integer registers
Arithmetic, logical	**Operations on integer or logical data in integer registers; signed arithmetic instructions trap on overflow**
ADD, ADDI, ADDU, ADDUI	Add, add immediate (all immediates are 16 bits); signed and unsigned
SUB, SUBI, SUBU, SUBUI	Subtract, subtract immediate; signed and unsigned
MULT, MULTU, DIV, DIVU	Multiply and divide, signed and unsigned; operands must be floating-point registers; all operations take and yield 32-bit values
AND, ANDI	And, and immediate
OR, ORI, XOR, XORI	Or, or immediate, exclusive or, exclusive or immediate
LHI	Load high immediate—loads upper half of register with immediate
SLL, SRL, SRA, SLLI, SRLI, SRAI	Shifts: both immediate (S__I) and variable form (S__); shifts are shift left logical, right logical, right arithmetic
S__, S__I	Set conditional: "__" may be EQ, NE, LT, GT, LE, GE
Control	**Conditional branches and jumps; PC-relative or through register**
BEQZ, BNEZ	Branch integer register equal/not equal to zero; 16-bit offset from PC
BFPT, BFPF	Test comparison bit in the FP status register and branch; 16-bit offset from PC
J, JR	Jumps: 26-bit offset from PC (J) or target in register (JR)
JAL, JALR	Jump and Link: save PC+4 to R31, target is 26-bit offset from PC (JAL) or a register (JALR)
TRAP	Transfer to operating system at a vectored address (see Chapter 5)
RFE	Return to user code from an exception; restore user mode (see Chapter 5)
Floating point	**Floating-point operations on DP and SP formats**
ADDD, ADDF	Add DP,SP numbers
SUBD,SUBF	Subtract DP,SP numbers
MULTD, MULTF	Multiply DP,SP floating point
DIVD, DIVF	Divide DP, SP floating point
CVTF2D, CVTF2I, CVTD2F, CVTD2I, CVTI2F, CVTI2D	Convert instructions: CVTx2y converts from type x to type y, where x and y are one of I (integer), D (double precision), or F (single precision); both operands are in the FP registers
__D, __F	DP and SP compares: "__" may be EQ, NE, LT, GT, LE, GE; sets comparison bit in FP status register

Notation	Meaning	Example	Meaning
←	Data transfer. Length of the transfer is given by the destination's length; the length is specified when not clear.	R1←R2;	Transfer contents of R2 to R1. Registers have a fixed length, so transfers shorter than the register size must indicate which bits are used.
M	Array of memory accessed in bytes. The starting address for a transfer is indicated as the index to the memory array.	R1←M[x];	Place contents of memory location x into R1. If a transfer starts at M[i] and requires 4 bytes, the transferred bytes are M[i], M[i+1], M[i+2], and M[i+3].
$\leftarrow_n$	Transfer an *n*-bit field, used whenever length of transfer is not clear.	M[y]$\leftarrow_{16}$M[x];	Transfer 16 bits starting at memory location x to memory location y. The length of the two sides should match.
X_n	Subscript selects a bit.	$R1_0\leftarrow 0$;	Change sign bit of R1 to 0. (Bits are numbered from MSB starting at 0.)
$X_{m..n}$	Subscript selects a bit field.	$R3_{24..31}\leftarrow$M[x];	Moves contents of memory location x into low-order byte of R3.
X^n	Superscript replicates a field.	$R3_{0..23}\leftarrow 0^{24}$;	Sets high-order three bytes of R3 to 0.
##	Concatenates two fields.	$R3\leftarrow 0^{24}$ ## M[x] F2##F3$\leftarrow_{64}$M[x];	Moves contents of location x into low byte of R3; clears upper three bytes. Moves 64 bits from memory starting at location x; first 32 bits go into F2, second 32 into F3.
*, &	Dereference a pointer; get the address of a variable.	P*←&x;	Assign to object pointed to by p the address of the variable x.
<<,>>	C logical shifts (left,right)	R1 << 5	Shift R1 left 5 bits.
==, !=, >, <, >=, <=	C relational operators: equal, not equal, greater, less, greater or equal, less or equal	(R1==R2) & (R3!=R4)	True if the contents of R1 equal the contents of R2 and the contents of R3 do not equal the contents of R4.
&, |, ^, !	C bitwise logical operations: and, or, exclusive or, and complement.	(R1 & (R2 | R3))	Bitwise and of R1 and the bitwise or of R2 and R3.

DLX Pipeline Structure

Stage	ALU instruction	Load or store instruction	Branch instruction

IF	IR←Mem[PC];
	PC←PC+4;
		IR←Mem[PC];
		PC←PC+4;
			IR←Mem[PC];
			PC←PC+4;

ID	A←Rs1; B←Rs2; PC1←PC
	IR1←IR
		A←Rs1; B←Rs2; PC1←PC
		IR1←IR
			A←Rs1; B←Rs2; PC1←PC
			IR1←IR

EX	ALUoutput←A *op* B;
	or
	ALUoutput←A *op* $((IR1_{16})^{16}\#\#IR1_{16..31})$;
		DMAR← A+ $((IR1_{16})^{16}\#\#IR1_{16..31})$;
		SMDR← B;
			ALUoutput←PC1 + $((IR1_{16})^{16}\#\#IR1_{16..31})$;
			cond←(Rs1 *op* 0);

MEM	ALUoutput1← ALUoutput	LMDR←Mem[DMAR]; or
		Mem[DMAR]←SMDR;
			if (cond) PC←ALUoutput;

WB	Rd←ALUoutput1;	Rd←LMDR;	