Do NOT Copy
From “Fundamentals of Computer Organization and Design”

Sivarama P. Dandamudi
School of Computer Science
Carleton University

September 24, 2003

Chapter 1

Vector Processors

Vector machines are special-purpose systems targeted for high-performance scientific computations in which matrix and vector arithmetic operations are quite common. What is a vector? It is a linear array of numbers: integers or floating-point numbers. From a programming language perspective, a vector is a one-dimensional array. That does not mean that vector processors work only one-dimensional arrays. As we show later in this section, we can treat a column of a two-dimensional matrix as a vector.

1.1 What Is Vector Processing?

Vector machines are designed to operate at the vector level. To illustrate vector operations, let's assume that we have two vectors **A** and **B**, each with 64 elements. The number of elements in a vector is called the vector size. In our example, the vector size is 64. We write our vectors **A** and **B** as

$$\mathbf{A} = a_0, a_2, \ldots, a_{n-1}\,,$$

$$\mathbf{B} = b_0, b_2, \ldots, b_{n-1}\,.$$

Suppose that we want to add these two vectors and place the result in vector **C**. Note that adding two vectors involves adding the corresponding elements as shown below:

$$\mathbf{C} = \mathbf{A} + \mathbf{B} = a_0 + b_0, a_1 + b_1, \ldots, a_{n-1} + b_{n-1}\,.$$

In high-level programming languages, we perform the vector addition by using a loop that iterates n times, where n is the vector size. In the C language, we can write this code as

```
for(i=0; i<n; i++)
    C[i] = A[i] + B[i];
```

This `for` loop iterates n times. The addition operation in this code is called the *scalar operation*. Vector processor instructions specify vector operations. For example, we just need one vector instruction to add vectors **A** and **B**. A typical vector instruction specifies four fields—three registers and an operation:

VOP	Vd	Vs1	Vs2

which performs `Vd = Vs1 VOP Vs2`. As you can see from this format, it uses the three-address format. Here is what a vector addition instruction **V**3 = **V**2 + **V**1 looks like in the Cray X-MP assembly language:

```
V3    V2 + V1
```

We need some more details before we can use vector instructions for **A** + **B**. We discuss the necessary Cray machine details later. As this example illustrates, a single vector instruction replaces the entire loop. There is no need to increment the index variable i and test it for the boundary condition (in our example, whether it is less than 64) to jump back. Due to this and their exploitation of pipelining, vector processors give superior performance for scientific computations.

1.2 Architecture

A vector machine consists of a scalar unit and a vector unit. The scalar unit works on scalars and has an architecture similar to that in the traditional processors. The vector unit is responsible for performing vector operations. Similar to the move from CISC to RISC designs, vector architectures have also progressed from the memory–memory architecture to the vector–register architecture.

The first vector machines used the memory–memory architecture. In this architecture, all vector operations receive the input operands from memory and store the result in memory. The first vector machine CDC Star 100 used this architecture.

Most of the current vector machines including the machines from Cray, NEC, Fujitsu, Hitachi, and Convex use the vector–register approach. This architecture is analogous to the RISC approach. All vector operations are done on vectors located in vector registers. The result is stored in a vector register as well. As with the RISC processors, special load and store instructions move vectors between vector registers and memory.

Architecture of a vector–register machine is shown in Figure 1.1, which is based on the Cray 1 system. It consists of five components: vector registers, scalar registers, vector functional units, vector load/store unit, and main memory.

Vector Registers: We have already mentioned the need for vector registers to hold the input and result vectors. The Cary 1 and most vector processors have eight vector registers. Each register can hold 64 elements of 64 bits each.

Some processors such as the Fijitsu VP 200 allow the total available register space of 8 K elements into a programmable set of vector registers ranging from 8 to 256. When using eight vector registers, each register can hold 1024 64-bit elements; at the other end, with 256 registers, each can hold only 32 elements.

As shown in Figure 1.1, each register has two read ports and one write port to allow a high degree of overlap among vector operations on different vector registers.

Scalar Registers: The scalar registers provide scalar input to vector operations. For instance, when multiplying all elements of a matrix by a constant, a scalar register supplies the constant. For example, to compute

B = 5 * **X** + **Y**

the constant 5 is stored in a scalar register, and vectors **X** and **Y** are stored in two vector registers. The scalar registers are also used to compute addresses for the vector load/store unit.

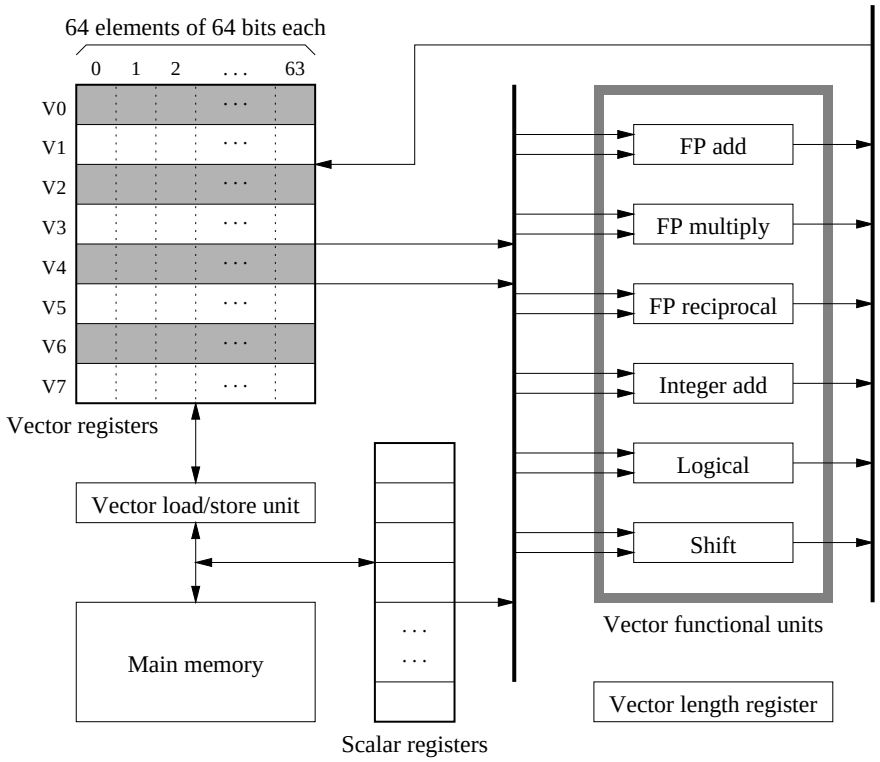


Figure 1.1 A typical vector processor architecture (based on the Cray 1).

Vector Load/Store Unit: This unit is responsible for movement of data between vector registers and memory. This unit is pipelined to facilitate overlapped read and write operations from memory. As well, it will mask the high latency associated with main memory access. This unit supports two basic operations: a load vector operation to move a vector from memory to a vector register and a store vector to move a vector to the memory. One load/store unit is typical, however, several vector processors have more than one such unit. For example, the NEC SX/2 has eight vector load/store units.

Vector Functional Units: Vector processors contain several vector functional units for floating-point, integer, and logical operations. For example, the Cray 1 has 6 functional units as shown in Figure 1.1. The NEC SX/2 has 16 functional units: 4 integer add/logical, 4 FP multiply/divide, 4 FP add, and 4 shift units. Note that these processors have other functional units for scalar operations, which are not our focus here.

Memory: The memory unit is different in organization than the ones we use in other processors. The memory organization should allow pipelined transfer of data to and from memory. Interleaved mem-

ory is used to support pipelined transfer of data from memory. We describe interleaved memories in Section 1.10 (see page 15).

1.3 Advantages of Vector Processing

Even though vector processing is not for general-purpose computation, it does offer significant advantages for scientific computing applications. To illustrate the performance advantages offered by vector systems, let's look at the simple vector addition example we have discussed before.

- *Flynn's bottleneck* can be reduced by using vector instructions as each vector instruction specifies a lot of work. As we have seen in the previous example, vector instructions can specify the work equivalent of an entire loop. Thus, fewer instructions are required to execute programs. This reduces the bandwidth required for instruction fetch.
- *Data hazards* can be eliminated due to the structured nature of the data used by vector machines. We can determine the absence of a data hazard at compile-time, which not only improves performance but also allows for planned prefetching of data from memory. The next point elaborates on this advantage.
- *Memory latency* can be reduced by using pipelined load and store operations. For example, when we fetch the first element in a 64-element addition operation, we can schedule fetching the remaining 63 elements from memory. By using interleaved memory designs, vector machines can amortize the high latency associated with memory access over the entire vector. We discuss interleaved memories on page 15.
- *Control hazards* are reduced as a result of specifying a large number of iterations in a single vector instruction. The number of iterations depends on the size of the vector registers. For example, if the machine has 64-element vector registers, each vector instruction can specify work equivalent to 64 loop iterations.
- *Pipelining* can be exploited to the maximum extent. This is facilitated by the absence of data and control hazards. Vector machines not only use pipelining for integer and floating-point operations but also to feed data from one functional unit to another. This process, known as chaining, is discussed on page 11. In addition, as mentioned before, load and store operations also use pipelining.

As you can see from this discussion, vector machines restrict the type of data to vectors, which exhibit known access patterns and reduce data and control hazards. This characteristic of the input data is heavily exploited by using pipelining in every aspect of the system operation.

1.4 The Cray X-MP

The first prototype of the Cray X-MP was introduced in 1982. It can support up to four CPUs. The processors communicate with each other through shared communication register clusters. In this section, we present an overview of its CPU architecture. For more details, see [2].

The Cray X-MP shares many of the RISC characteristics. Vector processors use the load/store architecture. In this architecture, all operations are done on operands that are in either scalar or vector

registers. Special load and store instructions are responsible for moving data between main memory and the processor registers.

As the Cray machine is targeted for scientific computations, all data in memory are 64 bits wide. It uses 22-bit addresses, where each address specifies a 64-bit word. Instructions are encoded into either a 16- or 32-bit format. In the Cray's terminology, a 16-bit instruction encoding is called *one parcel*, and the 32-bit encoding is *two parcels*.

The Cray X-MP has three types of registers: *address*, *scalar*, and *vector registers*. It has eight 24-bit address registers A0 to A7. These registers are mainly used for holding memory addresses for load and store operations. The use of 24-bit address registers may seem strange as the X-MP uses 22-bit addresses. The reason is that up to four instructions can be at the addressed location. Thus, we need two additional bits to identify one of the four instructions that could be packed into a 64-bit word.

The Cray X-MP uses pipelined execution for address, scalar, and vector operations. It has several address, scalar, and vector functional units. The address section of the processor has two functional units to perform address arithmetic operations. Details about the address functional units are shown below:

Address functional unit	Number of stages
24-bit integer ADD	2
24-bit integer MULTIPLY	4

The Cray provides several instructions to manipulate addresses. Here are some example address instructions in the Cray assembly language (CAL) format:

```
Ai  Aj+Ak    (Ai = Aj+Ak)
Ai  Aj*Ak    (Ai = Aj*Ak)
Ai  Aj−Ak    (Ai = Aj−Ak)
Ai  −Ak      (Ai = −Ak)
Ai  Aj+1     (Ai = Aj+1)
Ai  Aj−1     (Ai = Aj−1)
Ai  expr     (Ai = expr)
```

The last instruction can be used to assign constant values (`expr`) to A registers. As we show later, address registers are also used for holding short integers. For example, address registers are used to hold the shift count for shift instructions.

The scalar section of Cary X-MP has eight 64-bit scalar registers (S0 to S7) and four types of functional units: ADD, LOGICAL, SHIFT, and POP/PARITY. The first three operations are the standard addition, logical, and shift operations. The POP/PARITY unit counts the number of ones or zeros or determines parity. Table 1.1 shows the number of stages used in each scalar functional unit.

Some example scalar instructions are given below:

```
Si  Sj+Sk    (Si = Sj+Sk)
Si  Sj!Sk    (Si = Sj OR Sk)
Si  Si>Ak    (Si = Si>>Ak)
```

Table 1.1 Sample Cray X-MP scalar functional units

Scalar functional unit	Number of stages
Integer ADD (64-bit)	3
64-bit SHIFT	2
128-bit SHIFT	3
64-bit LOGICAL	1
POP/PARITY (population or parity)	4
POP/PARITY (leading zero count)	3

Table 1.2 Sample Cray X-MP vector functional units

Vector functional unit	Number of stages	Available to chain	Vector results
Integer ADD (64-bit integer)	3	8	VL + 8
64-bit SHIFT (64-bit logical)	3	8	VL + 8
128-bit SHIFT (128-bit logical)	4	9	VL + 9
Full vector 64-bit LOGICAL	2	7	VL + 7
Second vector 64-bit LOGICAL	4	9	VL + 9
POP/PARITY	5	10	VL + 10
Floating ADD	6	11	VL + 11
Floating MULTIPLY	7	12	VL + 12
Reciprocal approximation	14	19	VL + 19

The first instruction places the sum of `Sj` and `Sk` in the `Si` register. The second instruction performs the logical bitwise OR operation, and the third one right-shifts the contents of `Si` by `Ak` positions.

Like the Cray-1 shown in Figure 1.1, the X-MP has eight 64-element vector registers V0 to V7. Each vector register can hold 64 words (each word is 64 bits wide). Table 1.2 summarizes the details of the vector functional units. For now, ignore the last two columns in this table.

A sample of the Cray X-MP vector instructions is shown in Table 1.3. Each vector instruction works on the first VL elements. The VL value is in the vector length register (see Figure 1.1). The first group of instructions in this table shows the types of addition instructions provided by the X-MP. Similar instructions are available for multiplication and subtraction operations. Multiplication and subtraction instructions use * and − operators, respectively. There is no division operation. Instead, X-MP provides the reciprocal operation as shown in Table 1.3. We have shown only the logical AND operation in this table. The Cray X-MP also provides instructions to perform logical OR (!) and XOR (\) operations.

Table 1.3 Sample Cray X-MP instructions

Instruction	Meaning	Description
`Vi Vj+Vk`	`Vi = Vj+Vk` Integer add	Add corresponding elements (in the range 0 to VL − 1) from `Vj` and `Vk` vectors and place the result in vector `Vi`
`Vi Sj+Vk`	`Vi = Sj+Vk` Integer add	Add the scalar `Sj` to each element (in the range 0 to VL−1) of `Vk` vector and place the result in vector `Vi`
`Vi Vj+FVk`	`Vi = Vj+Vk` Floating-point add	Add corresponding elements (in the range 0 to VL − 1) from `Vj` and `Vk` vectors and place the floating-point result in vector `Vi`
`Vi Sj+FVk`	`Vi = Sj+Vk` Floating-point add	Add the scalar `Sj` to each element (in the range 0 to VL − 1) of `Vk` vector and place the floating-point result in vector `Vi`
`Vi ,A0,Ak`	Vector load with stride `Ak`	Load into elements 0 to VL − 1 of vector register `Vi` from memory starting at address `A0` and incrementing addresses by `Ak`
`Vi ,A0,1`	Vector load with stride 1	Load into elements 0 to VL − 1 of vector register `Vi` from memory starting at address `A0` and incrementing addresses by 1
`,A0,Ak Vi`	Vector store with stride `Ak`	Store elements 0 to VL − 1 of vector register `Vi` in memory starting at address `A0` and incrementing addresses by `Ak`
`,A0,1 Vi`	Vector store with stride 1	Store elements 0 to VL − 1 of vector register `Vi` in memory starting at address `A0` and incrementing addresses by 1
`Vi Vj&Vk`	`Vi = Vj&Vk` Logical AND	Perform bitwise-AND operation on corresponding elements (in the range 0 to VL − 1) from `Vj` and `Vk` vectors and place the result in vector `Vi`
`Vi Sj&Vk`	`Vi = Sj&Vk` Logical AND	Perform bitwise-AND operation on 0 to VL − 1 elements of `Vk` and scalar `Sj` and place the result in vector `Vi`
`Vi Vj>Ak`	`Vi = Vj>>Ak` Right-shift by `Ak`	Right-shift 0 to VL − 1 elements of `Vj` by `Ak` and place the result in vector `Vi`
`Vi Vj<Ak`	`Vi = Vj<<Ak` Left-shift by `Ak`	Left-shift 0 to VL − 1 elements of `Vj` by `Ak` and place the result in vector `Vi`

1.5 Vector Length

In our discussion so far, we have not said anything about the size of the actual vector. We conveniently assumed that the size of the vector register is equal to the size of the vector we have. What happens if this is not true? In particular, we have to handle two cases: the vector size is less than the vector register size, and the vector size is larger than the vector register size. For concreteness, we assume 64-element vector registers as provided by the Cray systems. We first look at the simpler of these two problems.

1.5.1 Handling Smaller Vectors

If our vector size is smaller than 64, we have to let the system know that it should not operate on all 64 elements in the vector registers. This is fairly simple to do by using the vector length register. The VL register holds the valid vector length. All vector operations are done on the first VL elements (i.e., elements in the range 0 to VL $-$ 1). There are two instructions to load values into the VL register:

```
VL   1        (VL = 1)
VL   Ak       (VL = Ak where k≠0)
```

For example, if the vector length is 40, the following code can be used to add two vectors in registers V3 and V4:

```
A1   40       (A1 = 40)
VL   A1       (VL = 40)
V2   V3+FV4   (V2 = V3 + V4)
```

Since we cannot write

```
VL   40
```

we have to use the two-instruction sequence to load 40 into the VL register. The last instruction specifies floating-point addition of vectors V3 and V4. Since the VL is 40, only the first 40 elements are added.

1.5.2 Handling Larger Vectors

The VL register is useful for handling smaller vector sizes, but it does not help for larger vector sizes. Suppose we have 200-element vectors (i.e., $N = 200$). How do we use the vector instructions to add two such vectors? The case of larger vectors is handled by a technique known as *strip mining*.

In strip mining, the vector is partitioned into strips of 64 elements. This leaves one odd-size piece that may be less than 64 elements. The size of this piece is given by (N mod 64). We load each strip into a vector register and apply the vector addition instruction. The number of strips is given by $(N/64) + 1$. For our example, we divide the 200 elements into four pieces: three pieces with 64 elements and one odd piece with 8 elements. We use a loop that iterates four times: in one of the iterations, we set VL to 8 and the remaining three iterations will set the VL register to 64.

1.6 Vector Stride

To understand vector stride, we have to consider how the elements are stored in memory. Let's first look at vectors. Since vectors are one-dimensional arrays, storing a vector in memory is straightforward:

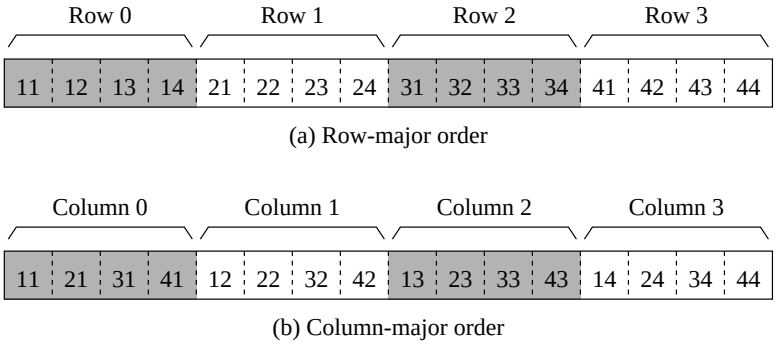


Figure 1.2 Memory layout of vector **A**.

vector elements are stored as sequential words in memory. If we want to fetch 40 elements, we read 40 contiguous words from memory. These elements are said to have a stride of 1. That is, to get to the next element, we add 1 to the current element. Note that the distance between successive elements is not measured in bytes; rather in number of elements.

We will need nonunit vector strides for multidimensional arrays. To see why, let's focus on two-dimensional matrices. If we want to store a two-dimensional matrix in memory, we have to linearize it. We can do this in one of two ways: *row-major* or *column-major* order. Most languages, except FORTRAN, use the row-major order. In this ordering, elements are stored in row order: row 0, row 1, row 2, and so on. In the column-major order, which is used by FORTRAN, elements are stored column by column: column 0, column 1, and so on. As an example, consider the following 4×4 matrix:

$$\mathbf{A} = \begin{bmatrix} 11 & 12 & 13 & 14 \\ 21 & 22 & 23 & 24 \\ 31 & 32 & 33 & 34 \\ 41 & 42 & 43 & 44 \end{bmatrix}.$$

This matrix is stored in memory as shown in Figure 1.2.

Assuming row-major order for storing, how do we access all elements of column 0? Clearly, these elements are not stored contiguously. We have to access 0, 4, 8, and 12 elements in the memory array. Since successive elements are separated by 4 elements, we say that the stride is 4. Vector machines provide load and store instructions that take the stride into account. We can see from Table 1.3 that X-MP supports both unit and nonunit stride access. For example, the instruction

```
Vi   ,A0,Ak
```

loads vector register `Vi` with stride `Ak`. Since unit stride is very common, a special instruction

```
Vi   ,A0,1
```

is provided. Similar instructions are available for storing vectors in memory.

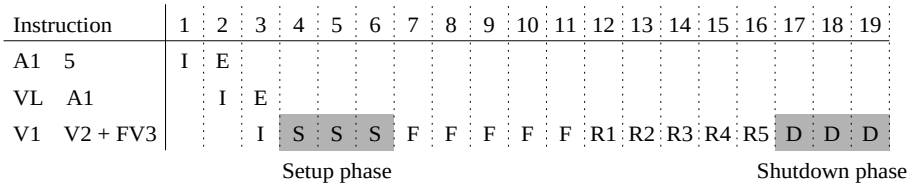


Figure 1.3 Timing analysis of a simple vector addition operation.

1.7 Vector Operations on the Cray X-MP

All vector units produce a result on each clock once the pipeline is full. Pipelined operations on the Cray X-MP proceed in three phases: *setup*, *execution*, and *shutdown*. During the setup phase, the functional unit is programmed to perform the appropriate operation and the connections to the source and destination vector registers are established. The setup time for all vector units is three clock cycles. The pipeline performs the computation during the execution phase. The number of cycles required to complete the computation depends on the vector length and the number of stages in the functional unit. The final shutdown phase represents the time difference between when the final result comes out of the pipeline and when the destination register can be used for another operation. For the X-MP, the shutdown phase also takes three clock cycles.

With this information, it is relatively straightforward to estimate the number of clock cycles required to execute a piece of code. We illustrate the process by means of examples.

Example 1.1 *A simple vector add operation.*
As a first example, consider the following code that adds the first five elements of vectors **V**2 and **V**3 and places the result in **V**1:

```
A1   5
VL   A1          (VL = 5)
V1   V2+FV3
```

The timing for this code is shown in Figure 1.3. The first instruction takes two clock cycles: one for instruction fetch and the other for execution. The second instruction fetch is done during the second clock cycle. It also takes two clocks. The vector instruction is fetched in clock cycle 3. This is followed by a three-cycle setup phase (shown by S). The floating-point add unit is a six-stage pipeline (see Table 1.2). Thus, for the next five clocks, the pipeline gets filled (shown by F). The first result comes out during clock 12. We get one result out for the next four clock cycles. The last three cycles are for the shutdown (D) phase. □

From this example, we can see that the add operation is completed in 16 clocks, excluding the instruction fetch. That is, 16/5 = 3.2 clocks per addition. We reduce the overhead by increasing the vector length. In general, the number of clocks required to add VL size vectors is

$$3 + 6 + (\mathrm{VL} - 1) + 3\,.$$

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
A1 5	I	E																			
VL A1		I	E																		
V1 V2 + FV3			I	S	S	S	F	F	F	F	F	R1	R2	R3	R4	R5	D	D	D		
V4 V5 * FV6				I	S	S	S	F	F	F	F	F	F	R1	R2	R3	R4	R5	D	D	D

Figure 1.4 Overlapped execution of floating-point addition and multiplication operations.

The first three cycles are for the startup phase. The six cycles are the number of stages in the add pipeline. Thus, by nine clocks, the result is out. The remaining (VL $-$ 1) results will require (VL $-$ 1) clocks as we get one result per clock. The last three cycles are for the shutdown phase. For example, if we use 64 element vectors, the number of clocks for each addition reduces to 75/64 $\approx$ 1.2 clocks.

By generalizing the previous expression to an n-stage pipeline functional unit, we get

$$\text{Number of clock required} = 3 + n + (\text{VL} - 1) + 3 = \text{VL} + n + 5\,.$$

This matches with the expression given in the last column of Table 1.2.

The next example shows how overlapped execution among multiple functional units improves the performance.

Example 1.2 *Overlapped execution of vector operations.*
In this example, we consider two vector operations that are independent. These instructions can be executed in any order.

```
A1   5
VL   A1          (VL = 5)
V1   V2+FV3      (Floating-point addition of V2 and V3)
V4   V5*FV6      (Floating-point multiplication of V5 and V6)
```

The timing for this code is shown in Figure 1.4. The first three rows are exactly the same as in the previous example. The floating-point multiply unit is a seven-stage pipeline. Thus, the first result is available during clock cycle 14. As in the add instruction, there are three clocks for the setup phase and three for the shutdown phase.

This example illustrates the overlapped execution of the floating-point add and multiply units. We get significant performance gains by this overlapped execution. Without overlapping these two operations, we would have required 16 clocks for the addition and 17 clocks for the multiplication. Using overlapped execution, we complete both operations in 18 clocks. □

1.8 Chaining

We have demonstrated in the last example that overlapped execution is important to get performance gains. Since there is no conflict on the vector registers used by the addition and multiplication operations,

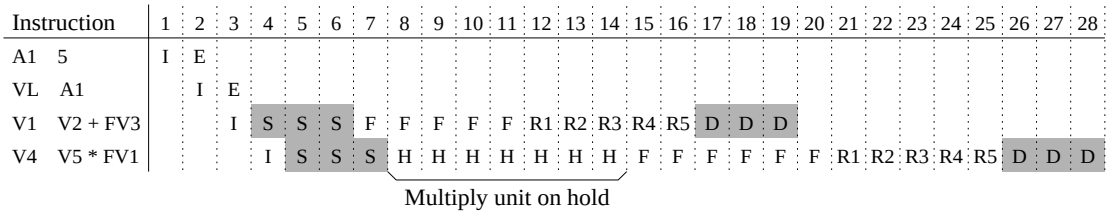


Figure 1.5 A vector chaining example.

we could overlap the execution of the two operations. What happens if there is a conflict as in the following example?

```
A1   5
VL   A1          (VL = 5)
V1   V2+FV3      (Floating-point addition of V2 and V3)
V4   V5*FV1      (Floating-point multiplication of V5 and V1)
```

The multiplication operation takes values from **V**1, which are produced by the addition operation. Due to this dependency, we cannot independently schedule these two instructions. As we have seen, sequential execution of these two instructions takes 16 + 17 = 33 clocks. This is where chaining is useful.

Chaining allows feeding of data from one operation to the next without waiting for the first operation to complete. For example, once the first result of addition is placed in **V**1, we could make that value available for the multiplication instruction. The Cray X-MP allows using the first result after two clock cycles of delay.

Example 1.3 *A chaining example.*
To understand how chaining speeds up execution, let's consider the example code given before. The execution timing is shown in Figure 1.5. The first three rows are the same as in the previous examples. The vector multiplication instruction uses chaining to get values from the **V**1 vector register. The addition operation produces the first result R1 during clock cycle 12. This result is available for use by the multiplication two clocks later. Until that time, the multiplication pipeline is in the hold (H) state. From that point onward, execution proceeds in the normal fashion. Thus, by using chaining, we complete both operations in 25 cycles. This is significantly faster than executing without chaining: 33 cycles versus 25 cycles, about 24% improvement. □

1.9 Performance

This section briefly discusses the performance of pipelined and vector processors.

1.9.1 Pipeline Performance

Performance of pipelined execution depends on the number of stages in the pipeline as well as the number of computations performed (e.g., vector length). Speedup is used to measure the relative performance of the pipelined system with the corresponding nonpipelined system. The

speedup is defined as

$$\text{Speedup} = \frac{\text{Nonpipelined execution time}}{\text{Pipelined execution}} .$$

Ideally, an n-stage pipeline should give a speedup of n. However, the pipeline would not operate at full capacity all the time due to the following:

- *Pipeline Fill:* When the computation is initiated, the pipeline is empty. It takes n cycles to fill an n-stage pipeline. Until that time, all stages of the pipeline are not busy.
- *Pipeline Drain:* At the trailing end of computation, no further input enters the pipeline. However, we have to drain the pipeline to get out all the results. Again, during this draining phase, all stages are not busy.

To compute speedup, let us assume that we perform N computations. Each computation takes $n \times T$ time units. Thus, on a nonpipelined system, it takes $N \times n \times T$ time. If we use an n-stage pipeline, we get the first result out in $n \times T$ time. But after that the pipeline is full and produces one result every T time units. Thus, we need an additional $(N-1)T$ time for the remaining $(N-1)$ results. Therefore, on a pipelined system, we need

$$n \times T + (N-1)T = (n + N - 1)T .$$

We can now compute the speedup as

$$\text{Speedup} = \frac{n \times N \times T}{(n+N-1)T} = \frac{n \times N}{n+N-1} .$$

By rewriting the above expression, we get

$$\text{Speedup} = \frac{1}{\frac{1}{N} + \frac{1}{n} - \frac{1}{n \times N}} .$$

We get the ideal speedup of n as $N \to \infty$. This condition implies that we are ignoring the "fill" and "drain" effects on the pipeline performance. At the other extreme, if $N = 1$, we get a speedup of 1, which agrees with our assertion that pipelining does not improve the execution time of a single computation. Figure 1.6*a* shows the speedup as a function of N in an n-stage pipeline.

This expression also tells us that we can improve the speedup by increasing the number of stages n. Instead of using an n-stage pipeline, we can design a $2n$-stage pipeline where each stage does half of the work done by one stage in the previous design. The speedup is than given by

$$\text{Speedup} = \frac{1}{\frac{1}{N} + \frac{1}{2n} - \frac{1}{2n \times N}} .$$

If we ignore other factors, we improve speedup as we increase the number of pipeline stages. The number of pipeline stages is also called the *pipeline depth*. This can be seen from Figure 1.6. For example, when we increase the pipeline depth n from 3 to 6, the speedup nearly doubles. Further increase in speedup is obtained by increasing the pipeline depth to 9 stages.

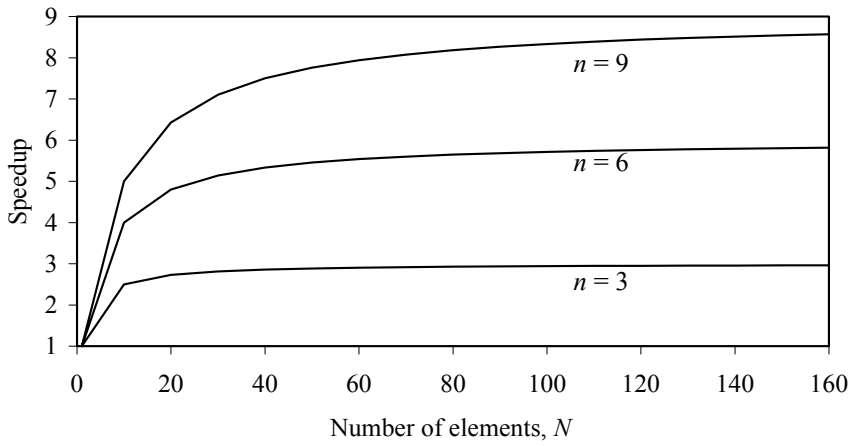


Figure 1.6 Pipeline performance as a function of the vector size N.

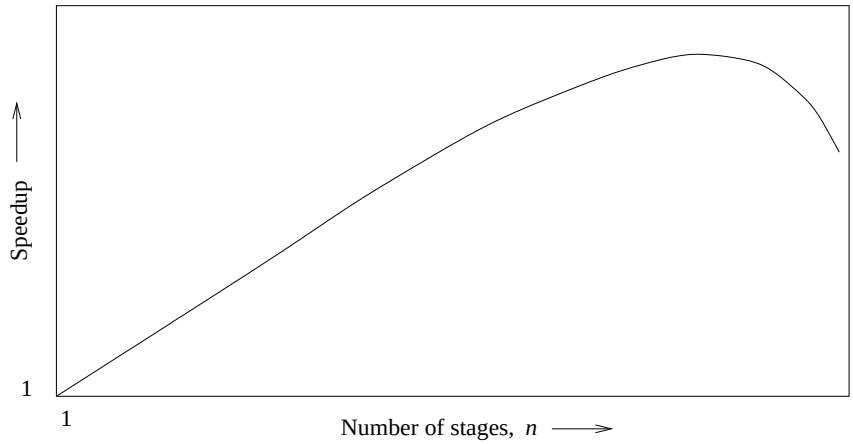


Figure 1.7 Pipeline speedup as a function of pipeline depth n.

It may appear from this that by increasing the pipeline depth, we can improve the performance. However, in practice, larger pipeline depth reduces performance (see Figure 1.7). This reduction is mainly contributed by the data and control hazards. The probability of a pipeline stall occurring due to these dependencies increases with the pipeline depth. Both these dependencies will cause the work in the pipeline to be thrown out. In addition, longer pipelines mean more time for branching when control hazards occur.

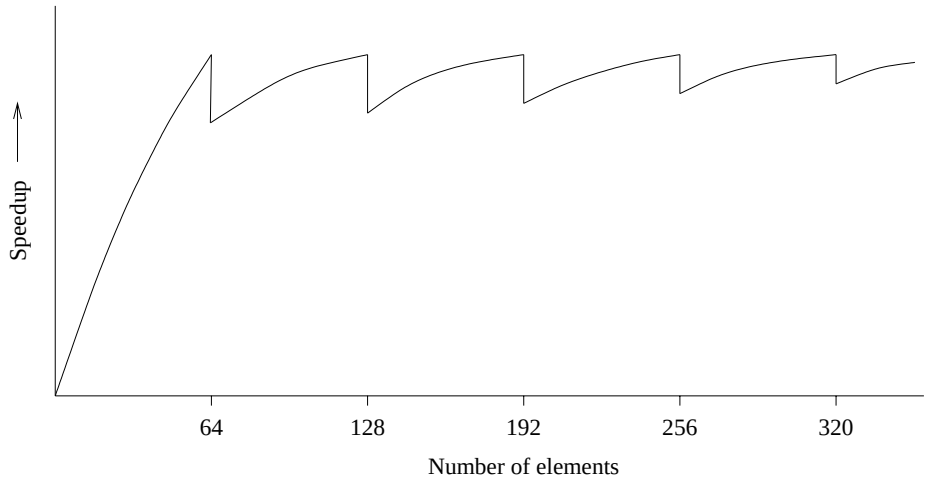


Figure 1.8 Vector processors exhibit "saw-tooth" shaped performance sensitivity to vector length.

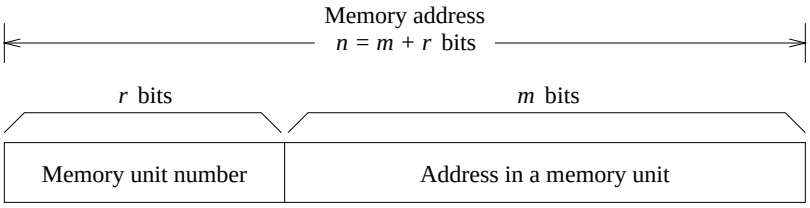
1.9.2 Vector Processing Performance

As we have seen in the Cray X-MP machine, vector systems use pipelining for address, scalar, vector, and load/store operations. Thus, the performance gains we get are based on the performance improvements due to pipelining. In this section, we look at the impact of vector register length on the performance of vector machines. As mentioned on page 8, we have to resort to strip mining to handle any mismatch between the actual vector and the vector register size VL.

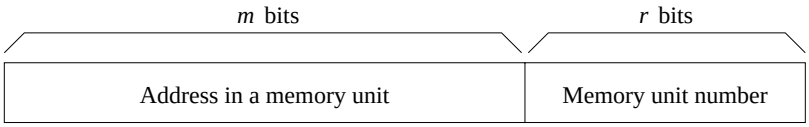
We get improved speedup as the vector size increases from 1 to VL. This is due to the amortization of the pipeline fill cost. As an example, consider a 10-stage pipeline with 64-element vector registers (VL = 64). Neglecting all other influences, nonpipelined operation on 64 elements takes 640 time units. On a pipelined unit, it takes 10 + 63 = 73 time units giving us a speedup of 640/73 = 8.77. Now look at what happens to the speedup when we increase the vector size by one element. On the nonpipelined unit, it takes 650 time units, whereas the pipelined unit takes 10 + 63 + 10 = 83. Thus, the speedup drops to 650/83 = 7.83. Similarly, you can verify that the speedup reaches 8.77 for 128-element vectors but drops to 8.27 for 129 elements. Thus, the performance peaks at multiples of VL but drops immediately after that, leading to the "saw-tooth" shaped performance shown in Figure 1.8. As with the pipeline performance, practical issues such as the time needed to load the vector registers would give much worse performance than that we can get from an ideal pipeline. For more details on pipeline and vector performance, see [1, 3, 2].

1.10 Interleaved Memories

In our memory designs thus far, we have mapped a block of contiguous memory addresses to a memory module. If we want a larger memory, we simply add more memory modules. This



(a) Normal memory address mapping



(b) Interleaved memory address mapping

Figure 1.9 Interleaved memories use the lower-order bits to select a memory bank.

modular design is useful in incrementally expanding the memory. This design, however, has one major disadvantage. Since sequential addresses are mapped to the same memory module, we have to wait for the memory access time for each word we read from the memory module. As an example, let's assume that memory access time is four clock cycles. Using our design, we need $8 \times 4 = 32$ cycles to read eight sequential words. Interleaved memories help reduce this access time.

1.10.1 The Concept

High-performance computers typically use interleaved memories to improve access performance. Essentially, these memories allow overlapped access to hide the memory latency. The key idea is to design the memory system with multiple banks and access all banks simultaneously so that access time can be overlapped. We explain this in detail next.

In the traditional memory designs, the n-bit memory address is divided into into two parts: the higher-order r bits are used to identify a memory module and the lower-order m bits are used to specify a location in the memory module. As shown in Figure 1.9a, $n = r + m$. This technique is sometimes called *high-order interleaving*.

To provide overlapped access to sequential addresses, we have to resort to *low-order interleaving*, as shown in Figure 1.9b. In this design, the lower-order r bits are used to identify a memory module and the higher-order m bits are used to specify a location. We normally use the term interleaved memory to mean low-order interleaving. In these designs, the memory modules are referred to as *memory banks*.

In interleaved memories, memory addresses are mapped on a round-robin basis. Thus, in a B-bank design, address 0 is mapped to bank 0, address 1 to bank 1, and so on. A memory

Table 1.4 Example mappings

Memory bank number	Memory addresses mapped
Bank 0	$0, B, 2B, \ldots$
Bank 1	$1, B+1, 2B+1, \ldots$
Bank 2	$2, B+2, 2B+2, \ldots$
. . .	. . .
Bank $B-1$	$B-1, 2B-1, 3B-1, \ldots$

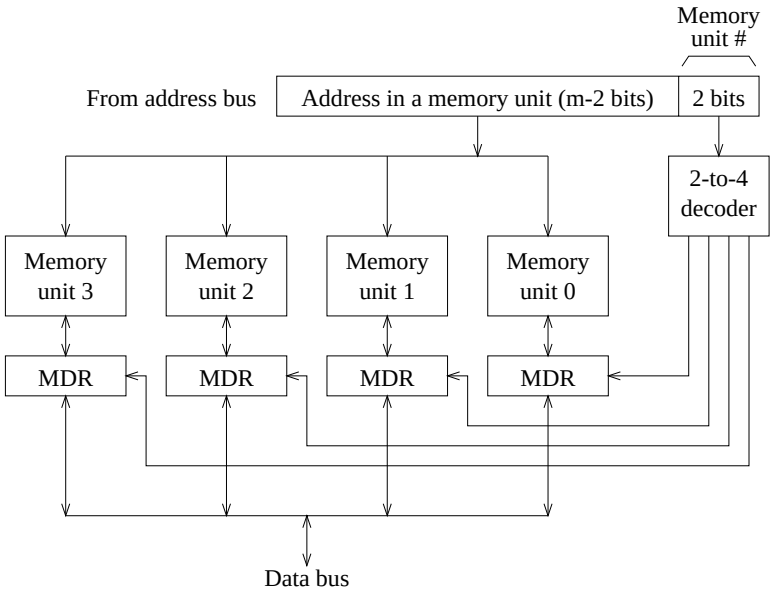


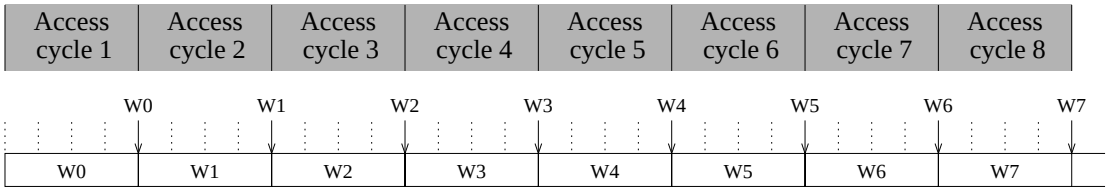
Figure 1.10 Interleaved memory design with synchronized access organization.

address $addr$ is mapped to memory bank $b = addr$ MOD B. Some example mappings are shown in Table 1.4.

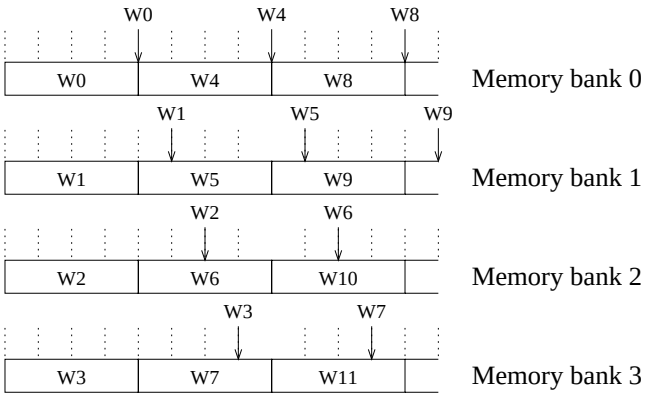
We can implement interleaved memories using two possible designs: synchronized access organization or independent access organization. These two organizations are described next.

1.10.2 Synchronized Access Organization

In this organization, the upper m bits are presented to all memory banks simultaneously. We illustrate the working of this design by means of an example. Figure 1.10 shows this design



(a) Noninterleaved memory access



(b) Interleaved memory access

Figure 1.11 Interleaved memory allows pipelined access to memory.

for a four-bank interleaved memory. As before, it is assumed that the memory access takes four clock cycles. Once the address is presented, all four memory banks initiate their memory access cycles. Thus, in four clock cycles, we will have four data words from the four banks. These data words are latched into four tristate memory data registers (MDRs). We can transfer these four words in four clocks by selecting the appropriate MDR from the lower-order 2-bits using a 2-to-4 decoder (see Figure 1.10). Simultaneously, we can present the next address to the memory banks to initiate the next memory access cycle. Thus, by the time we transfer the four words from the first access cycle, we will have the next four words from the second access cycle. This process is shown in Figure 1.11.

As shown in Figure 1.11a, noninterleaved memory access takes four clocks for each access. Interleaved memory, on the other hand, transfers the first word (W0) after four clocks (see Figure 1.11b). After that, one word is transferred every clock cycle. Thus, to transfer eight words, we need 12 clock cycles (as opposed to 32 clocks in a noninterleaved design).

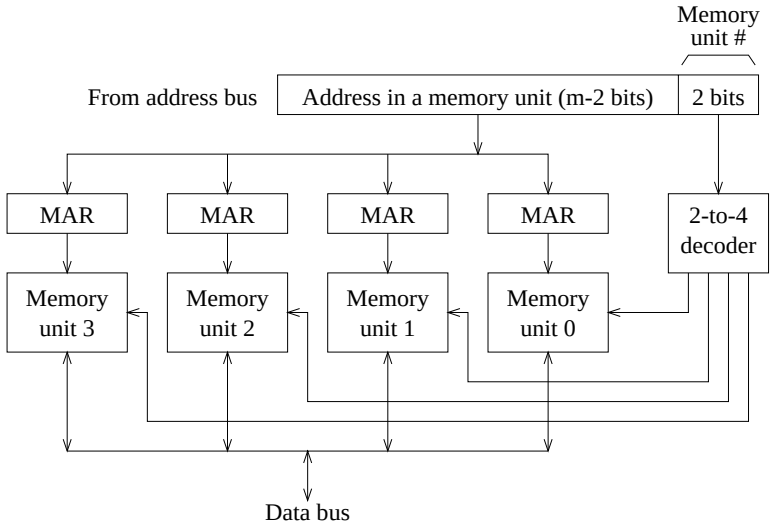


Figure 1.12 Interleaved memory design with independent access organization.

1.10.3 Independent Access Organization

A drawback with the synchronized design is that it does not efficiently support access to non-sequential access patterns. The independent access organization allows pipelined access even for arbitrary addresses. In order to support this kind of access, we have to provide each memory bank with a memory address register (MAR) to latch the address that the bank should use (see Figure 1.12). In our example, we can load four different addresses for the four banks. In this design, we do not need the MDR registers to latch the data. Instead, the data are read directly from the memory bank.

Independent design still provides the same kind of pipelined access as the synchronized access design. To see this, let's trace the first few data transfers from addresses 100, 81, 54, 31, 108, and 121. We can use (address MOD 4) to determine the memory bank number. The first address 100 is sent to bank 0. Once this address is latched in the MAR of bank 0, this bank can initiate the memory access cycle. During the next clock cycle, address 81 is latched into the MAR of bank 1. The next two addresses are sent to banks 2 and 3. After latching the address in the MAR of bank 3, data from bank 0 are available. So the next clock cycle is used to transfer the data from bank 0. We can now load the new address (108) in the MAR register of bank 0. Similarly, during the next clock, data from bank 1 are transferred, and a new address (121) is latched into the MAR of bank 1. This process repeats to provide pipelined access for arbitrary addresses. Of course, this organization also works for the sequential addresses.

1.10.4 Number of Banks

In our examples, we assumed four banks with a memory access time of four cycles. This gives us a data transfer rate of one word per cycle (after the initial startup delay). In general, how many memory banks do we need to provide one word per clock cycle transfer rate if the memory access time is M cycles? From our discussion, it should be clear that the number of banks B should be at least M.

We should caution the reader that interleaved memory does not always provide pipelined access to data. It depends on the memory access pattern. Next we give two examples representing best- and worst-case scenarios.

Example 1.4 *Suppose we have eight memory banks with an access time of six clock cycles. Find the time needed to read 16 words of a vector with stride 1.*

Note that a stride of 1 corresponds to sequential access, assuming that each address supplies a word (see page 8 for information on vector stride). This means we will be accessing the memory sequentially. Thus, whichever design we use, we will get pipelined access. This is the best-case scenario. Since the number of memory banks is at least six, we can get 1 word per cycle data transfer rate. Thus, to complete the transfer of 16 words, we have to wait 6 clocks for the first word and 16 additional clocks to transfer the 16 words. Therefore, we need $6 + 16 = 22$ clock cycles. □

Example 1.5 *Consider the memory system in the previous example. Find the time needed to read 16 words of a vector with stride 8.*

Since the stride is equal to the number of memory banks, we know that all 16 word addresses are mapped to the same bank. In this case, interleaving is not useful at all. We have to access each word sequentially from the same bank. Thus, we need $16 \times 6 = 96$ clock cycles. This is the worst-case scenario. □

1.10.5 Drawbacks

Why do we see interleaved memories only in high-performance computers? Why not in our PCs? One reason is that interleaved memories involve complex design. We have seen some of this complexity in our examples. We need extra registers (MAR or MDR) to latch addresses or data. What we have not shown in these examples is the control circuit needed to transfer data and to load addresses.

Another major reason is that interleaved memories do not have good fault-tolerance properties. That is, if one bank fails, we lose the entire memory. On the other hand, in traditional designs, failure of a module disables only a specific area of the memory address space mapped to the failed module. Furthermore, interleaved memories do not lend themselves well to incremental memory expansion.

1.11 Summary

We have demonstrated that pipelining can significantly improve the performance of a processor. The key factor is the overlapped execution of multiple operations. Although pipelining does not reduce the time needed to perform a single operation, it does increase the throughput using overlapped execution. Pipelining involves highly coordinated movement of data from stage to stage. Any delay in one stage can seriously affect the performance of the whole pipeline.

Vector processors exploit pipelining fully to get substantial performance improvement. These machines are best suited for scientific computations wherein the structured data allow pipelined designs for most operations including the load and store operations. These machines use vector registers that can hold a small vector. The strip mining technique can be used to process larger vectors. We have presented details about the Cray X-MP processor. Performance of pipelined and vector processors is briefly discussed in the last section.

In the last section, we presented details on interleaved memories, which are typically used in vector processors to facilitate pipelined access to memory.

Bibliography

[1] R.N. Ibbett and N.P. Topham, *Architecture of High Performance Computers* (Volume 1), Springer-Verlag, New York, 1989.

[2] K.A. Robbins and S. Robbins, *The Cray X-MP/Model 24: A Case Study in Pipelined Architecture and Vector Processing*, Lecture Notes in Computer Science 375, Springer-Verlag, New York, 1987.

[3] D. Sima, T. Fountain, and P. Kacsuk, *Advanced Computer Architectures: A Design Space Approach*, Addison-Wesley, New York, 1997.