A Taxonomy for Computer Architectures

David B. Skillicorn
Queen's University at Kingston

Flynn's classification of architectures does not discriminate clearly between different multiprocessor architectures. Since the number of multiprocessor architectures has increased substantially, it has become important to find a useful way to describe them—a way that distinguishes those that are significantly different while revealing the underlying similarities between apparently divergent designs.

This article presents a taxonomy for computer architectures that extends Flynn's, especially in the multiprocessor category. It is a two-level hierarchy in which the upper level classifies architectures based on the numbers of processors for data and for instructions and the interconnections between them. A lower level can be used to distinguish variants even more precisely; it is based on a state machine view of processors. I suggest why taxonomies are useful in studying architecture and show how mine applies to a number of modern architectures.

This taxonomy extends Flynn's classification of architectures to be more discriminating. In particular, the growing variety of multiprocessors can be categorized and related.

There has been a rapid growth in the number of proposed and constructed architectures over the past 10 years. Many of these have been highly parallel in nature because the applications envisaged demand increased performance that uniprocessor systems cannot continue to provide. The relationships between the large number of parallel architectures are difficult to appreciate because of the lack of a sufficiently rich taxonomy. Only one formal taxonomy, due to Flynn,[1] is in general use and it does not begin to differentiate the large number of possible multiprocessor architectures. Flynn classifies architectures by the number of instruction and data streams that they can process simultaneously. The categories are:

- single instruction, single data (SISD)
- single instruction, multiple data (SIMD)
- multiple instruction, single data (MISD)
- multiple instruction, multiple data (MIMD)

In this article, I present a classification scheme, or taxonomy, that extends Flynn's to make it more discriminating. It is based on a functional view of architecture and on information flow between units. I will show that this scheme classifies existing architectures well and also suggests new possibilities.

Taxonomies are important in classifying the world. Good taxonomies should group together those objects that are strongly related in important rather than unimportant ways. For example, biologists group whales with other mammals (warm-blooded, live-bearing) rather than with fish (swim in water) because their relationship to mammals is more important—if less obvious—than their relationship to fish.

Biologists have also shown that realistic taxonomies are hierarchically structured. My taxonomy is a two-level structure that subsumes Flynn's classification. Gross discriminations can be made at the higher level, with finer distinctions made at the second level.

Other classification schemes have been described in the literature. Feng's classifi-

0018-9162/88/1100-0046$01.00 © 1988 IEEE

cation is perhaps the best known.[2] It is a performance-oriented classification that describes the parallelism of the set of processors in a machine in terms of the number of bits that can be processed simultaneously. Machines are described by a pair, the first element giving the width of a word, and the second the depth—the number of words that can be operated on simultaneously. It does enable comparisons of performance between widely differing architectures, but precisely because it relates diverse architectures it does not highlight their differences.

More descriptive classifications also exist. An early classification of Reddi and Feurstel[3] uses physical organization, information flow, and representation and transformation of information as the basis for classification. Information flow can be a powerful and general way of describing architectures, but their other attributes are very implementation oriented. Another popular descriptive classification is due to Händler.[4] His classification describes architectures by giving the number of processors and how they can be pipelined together, and the width and depth of the arithmetic/logic units. While this works well for describing conventional vectorized machines, it is not clear how to generalize it to newer multiprocessor architectures.

Reasons for an architectural model. There are three reasons for classifying architectures.

The first is to understand what has already been achieved. Until the past two decades, almost all computer systems used a von Neumann architecture. Even when the underlying hardware began to contain a limited amount of parallelism (for example, systems such as the CDC6600), it was generally concealed from the user. However, since that time, the growth in systems with different kinds of parallelism has been explosive, and it is not at all clear which architectures have the best prospects for the future.

The second reason for having a classification of architectures is that it reveals possible configurations that might not have otherwise occurred to a system designer. Once existing systems have been classified, the gaps in the classification can suggest other possibilities. Of course, not all such possibilities will be improvements. They may already have been thought of and discarded as worthless. However, until the search space has been clearly delimited, there can be no assurance that all the possibilities have been examined.

The third reason for a classification scheme is that it allows useful models of performance to be built and used. I have already mentioned that a drive for greater performance lies behind almost all new architectural ventures. A good classification scheme should reveal why a particular architecture is likely to provide a performance improvement. It can also serve as a model for performance analysis.

Approaching classification. If we consider the whole question of computation and machines that provide it, it seems helpful to think at several different abstraction levels. At the highest, most abstract level, we can consider the *model of computation* being employed. Most computers still employ a model of computation called the von Neumann model. In this view of computation, primitive operations consist of the usual operations of mathematics: addition, multiplication, comparison, and so on (in suitably restricted domains). Primitive data, however, are kept in named locations, and correct computational behavior is enforced by the programmer who describes the exact order in which computations are to be performed. In fact, the execution order is over-constrained in the sense that there are other orderings of all but trivial programs that compute the same result; but there is no way to express this in the computational model.

The von Neumann model of computation seems natural to those whose programming experience was developed using languages that support it. However, it simply does not contain any concept of computing more than one thing at once (parallelism), nor does it contain any way in which the possibility of ordering computations dynamically can be expressed.

It is not surprising that the first new computation models developed were extensions to the von Neumann model, to which concepts such as communication primitives were added. Later, other, more unusual, computation models—such as dataflow and graph reduction—were developed.

As we move to a more detailed level, we can consider *abstract machines* as the implementation of models of computation. The mapping from model of computation to abstract machine is not necessarily a 1-to-1 mapping; some models of computation can be implemented by more than one type of abstract machine, although there is usually a better fit for some than others. An abstract machine captures the essence of a particular architecture form, without distinguishing different technologies, implementation sizes, or the like. For instance, the traditional von Neumann abstract machine can be evoked for most programmers by a summary such as program counter, addressing modes, condition codes, control unit, and arithmetic/logic unit. This abstract machine does not distinguish between the different von Neumann implementations that exist, nor does it distinguish between reduced-instruction-set computers and complex-instruction-set computers. It is nevertheless a useful level at which to consider architectures, and it is this level that forms my taxonomy's highest level.

At the next, more detailed, level we can consider *machine implementations.* This need not necessarily concern only the technology used to implement the machine. Rather, this level represents one definition of computer architecture, the picture of the machine presented to an assembly language programmer. (This definition is itself embedded in the world of von Neumann machines, since in many other architectures it is not clear what an assembly language programmer is, let alone what his or her view of the machine would be.) This level corresponds to the second level of my taxonomy. One could also envisage a level below this in which details of the physical implementation are used to distinguish between different processors.

I have not discussed the role of programming languages in this classification because, in some sense, the language used on a particular machine corresponds to the model of computation that it uses. If this is not the case, then there is a very bad "fit" between language and implementation and, since our goal is (implicitly at least) performance, this sort of situation can be ignored. Of course, when the von Neumann machine was all that there was, many different languages were used on the same machine. However, there is a growing trend towards building the architecture and the language together, and there are good reasons for that trend. Even for the von Neumann architecture, it is clear that Algol-like languages are a better "fit" than, say, Lisp.

Functions of a computer architecture. In my classification of computer architectures, I use four types of functional units from which any abstract machine can be constructed. These are

- An instruction processor, a functional unit that acts as an interpreter

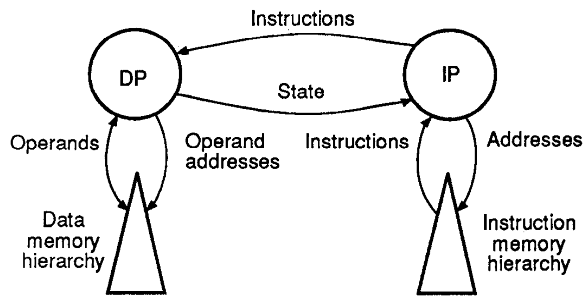

Figure 1. Arrangement of the von Neumann abstract machine functional units.

for instructions, when such things explicitly exist in the model of computation.

- A data processor, a functional unit that acts as a transformer of data, usually in ways that correspond to basic arithmetic operations.
- A memory hierarchy, an intelligent storage device that passes data to and from the processors.
- A switch, an abstract device that provides connectivity between other functional units.

In the next section, I illustrate how these functional units capture the behavior of an abstract machine, using the von Neumann architecture as an example.

The von Neumann abstract machine

The von Neumann abstract machine consists of a single instruction processor (IP), a single data processor (DP), and two memory hierarchies. The switch functional unit plays no role in describing the von Neumann abstract machine.

These functional units are arranged as shown in Figure 1. The instruction processor connects to one of the memory hierarchies from which it can fetch instructions. To do so, it must provide the memory hierarchy with a label (address) that identifies the instruction to be fetched. It also connects to the data processor, to which it sends information about operations to be performed and the labels of the values on which they are to be performed, and from which it receives state information (condition codes).

The functions of the instruction processor are to

(1) Determine the label of the next instruction to be executed, using information from its own internal storage and the state information conveyed to it by the data processor.
(2) Instruct the memory hierarchy to provide the instruction with that label.
(3) Receive the instruction and decode it. This involves determining which operation, if any, the data processor will be required to do to "execute" the instruction.
(4) Inform the data processor of the operation required.
(5) Determine the labels of the operands and pass them to the data processor.
(6) Receive the state information from the data processor (after completion of the operation).

Any architecture with an instruction processor will require it to carry out steps such as these. There is no presumption, in the description, that these steps need be carried out in the order given or that certain steps may not be skipped. At the first level of the taxonomy, only the existence of certain functional units, connected in a particular way, is specified.

For the von Neumann machine, the order of operations is usually that given above. It is called the instruction execution sequence. If we wish to indicate the precise order of the steps, then we move to the second level of the taxonomy and represent these various operations as states in a state diagram. At this second level, the instruction processor can be represented as in Figure 2, in which each state represents an operation and the arrows represent the dependencies between states. The dotted lines represent communication between the instruction processor and the other functional units in the system. If we visualize a "token" that can be placed on a state, then this state diagram—together with the token's location—gives a complete description of what the instruction processor is doing at any given moment. The token then follows the arrows between states, circling the loop once for each executed instruction.

We can continue to differentiate machines that operate according to the state diagrams given above by providing implementation level details. These might include exact mechanisms or timings for each state, or the way in which logical entities are mapped to physical ones. One such example is the following: Most von Neumann machines implement the selection of the next instruction using a program counter updated by the length of the current instruction (ignoring branch instructions). However, in some systems, the address of the next instruction is included in the instruction itself. Neither of these strategies is more von Neumann than the other. They are implementation choices and should be indistinguishable at the abstract machine level.

The data processor carries out the following steps:

(1) Receive an instruction type from the instruction processor.
(2) Receive operand labels from the instruction processor.
(3) Instruct the memory hierarchy (separate from that of the instruction processor) to provide the values of the operands.
(4) Receive operand values from the memory hierarchy.
(5) Carry out the required operation (the execute phase).
(6) Return state information to the instruction processor.
(7) Provide a result value to the memory hierarchy.

Once again, this can be represented by a state diagram consisting of a loop of states. The greatest difference between the instruction processors is that the execute phase of the data processor consists of a

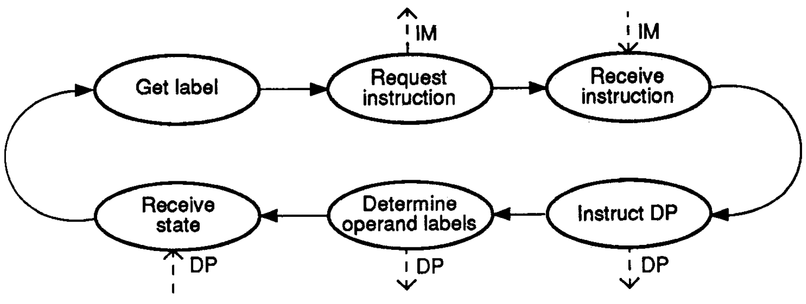

Figure 2. The internal structure of a von Neumann instruction processor.

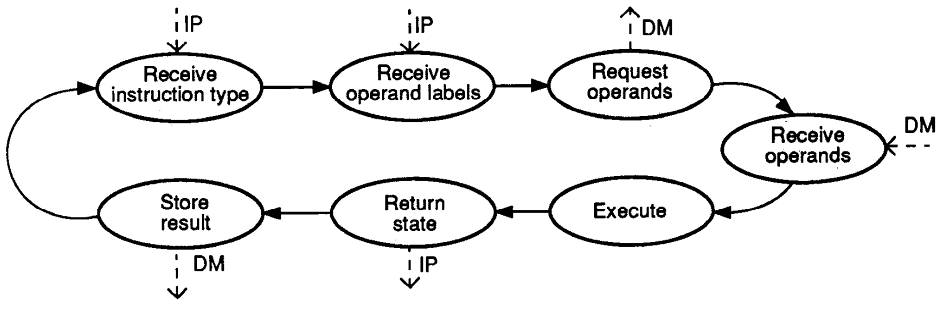

Figure 3. The internal structure of a von Neumann data processor.

potentially large number of parallel states, representing those operations that the data processor regards as primitive. The state diagram is shown in Figure 3.

Certain actions taken in both the instruction processor and data processor must be synchronized. Places where this occurs, shown using dotted lines in the state diagrams, represent communication between the processors. Their behavior follows: A token must be present on the state at the beginning and end of communication path for communication to occur. Neither token can leave that state until the communication is complete (that is, communication is synchronous).

A memory hierarchy is an intelligent storage device that attempts to provide the data requested by its attached processor as quickly as possible. I begin by justifying my treatment of a storage as a hierarchy. Consider the ultimate (at least with the physics we now understand) storage device. At its center is the processor. The data are stored in a sphere around it and can be accessed at light speed. As we increase the amount of storage, we are forced to place some storage locations on the outer surface of the sphere at a greater radius from the processor. Hence, the time to access those locations must increase. The surface area at the larger radius is greater as well, so more data can be stored there.

Therefore, we can see that storage must inherently be organized in a hierarchy. Data stored close to the processor will have short access times but must always be of limited capacity. Adding capacity forces an increase in access time for some storage. But storage with longer access times can be more plentiful. Thus, a storage system should be regarded as a pyramid or, more accurately, as a ziggurat, since the data stored are discrete and successive "layers" get larger in steps.

A memory hierarchy attempts to keep the next piece of data required by its attached processor at the top of the hierarchy where the access time is smallest. A perfect memory hierarchy would always achieve this goal. However, there is no optimal way to determine which piece of data will be required next. Hence, all memory hierarchies must approximate this goal using heuristics.

Clearly, the average access time is minimized when the data objects referenced most often are kept at the top of the hierarchy and those referenced least often

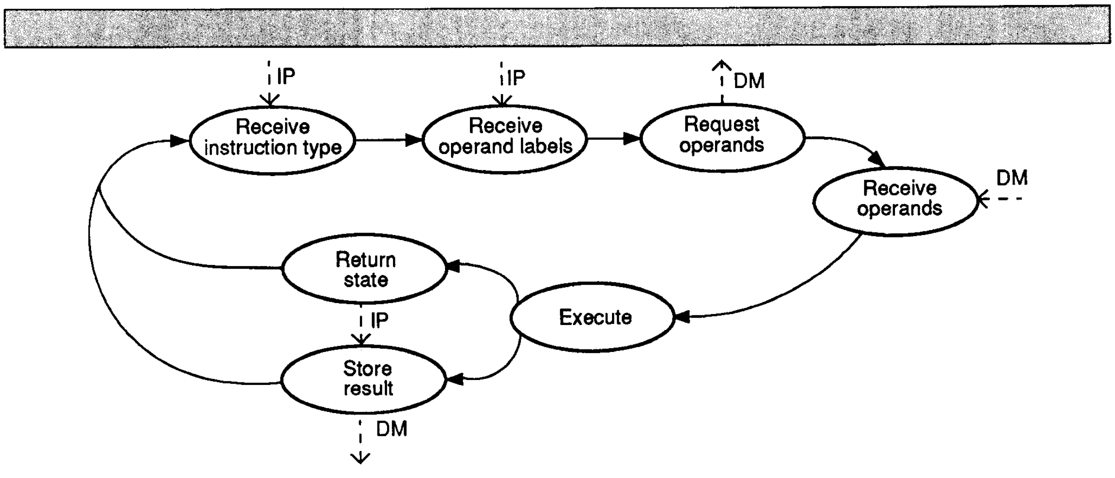

Figure 4. A faster von Neumann data processor.

are kept at the bottom. When the data accesses have some locality, either temporal or spatial, average access performance can be improved by keeping data that have been accessed recently or that are close to accessed data at the top of the hierarchy. Fortunately, almost all computation models do appear to exhibit substantial locality.

We can now see why a von Neumann abstract machine contains two memory hierarchies. Data in the instruction memory hierarchy flows towards the processor; instructions are absorbed by the instruction processor and do not themselves get modified. On the other hand, the data memory hierarchy has to manage data movement in both directions. Hence, the implementation of the two memory hierarchies may have substantially different properties. In principle, a von Neumann architecture does not distinguish between data and instructions; but, almost all real machines make it difficult for a programmer to use this equivalence, and programming environments enforce a separation. It seems useful to make the distinction in the abstract machine. Memory hierarchies are usually implemented using a single hierarchy of storage except at the top level—the cache—in which instruction and data storage are often differentiated. The cache is at the top of the memory hierarchy, above the main memory, which in turn is above disk storage (such as swap space) and longer term storage such as tapes. I/O takes place at the bottom of the memory hierarchy. In a sense, I/O is an access to a very large storage mechanism, the outside world.

Ways to increase performance

Most of the other computational models are motivated by a desire for increased performance, so it is instructive at this point to consider how the von Neumann abstract machine could achieve better performance. There are three major ways: rearranging the state diagrams so that it takes less time for a token to circulate; allowing more than one token and, hence, more than one active step at a time; and, replicating functional units to permit concurrent activity. All three ways do increase performance, and the third is especially fruitful, leading to a large and diverse class of architectures. Notice how explicit these possible mechanisms are made by the abstract machine representation.

The first method of increasing speed—rearranging the state diagram—does not really generate a new architecture class, although my classification scheme does allow differences to be distinguished in the descriptions of the processors' state diagrams. As an example, in the data processor, the operations of informing the instruction processor of the state information derived from the result and of storing the result in the memory hierarchy are independent. They can, therefore, occur simultaneously. This leads to the revised state diagram shown in Figure 4, which shows that, regardless of the time each step takes, the new arrangement will be faster.

Pipelining. The second possible speedup, permitting more than one token in the state diagram, is called pipelining. Suppose that the number of stages in a state diagrams is n. If each stage takes the same amount of time, say t, then the time to execute one instruction is $n\ t$. If we allow n tokens to be present in the state diagram simultaneously, we represent n different instructions in various stages of execution. The time it takes to execute each instruction is still $n\ t$. However, the average rate at which instructions are completed is $1/t$ instructions per unit of time. Thus, we have achieved an n-fold speedup in our machine, as long as there are always n tokens in the state diagram. Of course, there are practical difficulties in making this work—instructions such as branches and jumps, external interrupts, and simultaneous needs to access the same data cause real pipeline performance to be much worse than this simple analysis would suggest. However, it does serve to show why pipelines were interesting to system designers trying to improve the performance of von Neumann machines.

Pipelines have another interesting property. If we subdivide each of the states in our state diagram into two substates, each of which takes time $t/2$, and allow $2n$ tokens in the state diagram, then an instruction completes every $t/2$ time units thus increasing the completion rate of instructions to $2/t$ instructions per unit of time. Thus, we have a total speedup of $2n$. Making each stage smaller allows us to increase the completion rate, but at the expense of complicating the handling of unforeseen interactions.

Within our abstract machine model, pipelined behavior can be described without adding any new functional units. Instead, we label each functional unit as one of two types: simple or pipelined.

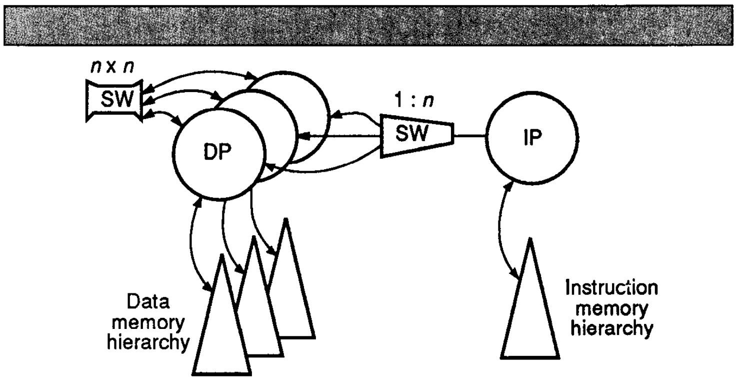

Figure 5. A Type 1 array processor.

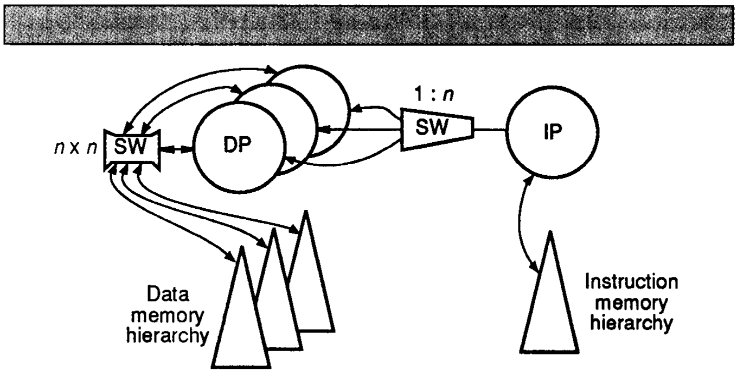

Figure 6. A Type 2 array processor.

Array processors. Replicating functional units is the third way to improve system performance. There are many different possibilities for replication, and we begin with the simplest—array processors. A typical array processor consists of an instruction unit that broadcasts instructions to a set of slave processors. Each of the slave processors executes the broadcast instruction, interpreting the operand addresses as lying in its own memory. Instructions are usually provided to allow processors to exchange data, directly or indirectly. It is also common for each processor to have access to its own identity, allowing for different relative locations to be accessed in different processors. An array processor is straightforwardly classified as an abstract machine with a single instruction processor, a single instruction memory, multiple data processors, and multiple data memories.

Because there are multiple functional units, we must describe the ways in which they can be connected. Connections between functional units are made using abstract switches that can be implemented in different ways: by buses, dynamic switches, or static interconnection networks. Four different forms of abstract switch connect functional units together:

- 1-to-1—A single functional unit of one type connects to a single functional unit of another. Of course, there may be more than one physical connection and information may flow in both directions, as in the von Neumann abstract machine described earlier. This is not really a switch, but I include it for completeness.
- n-to-n—In this configuration, the ith unit of one set of functional units connects to the ith unit of another. This type of switch is a 1-to-1 connection replicated n times.
- 1-to-n—In this configuration, one functional unit connects to all n devices of another set of functional units.
- n-by-n—In this configuration, each device of one set of functional units can communicate with any device of a second set and vice versa.

All array processors have a 1-to-n switch connecting the single instruction processor to the data processors. Two different subfamilies can be distinguished, based on the types of switch used to connect the data processors and data memory hierarchies. The first kind is shown in Figure 5. Here, the data processor-data memory connection is n-to-n and the data processor-data processor connection is n-by-n. This connection scheme is used in machines such as the Connection Machine.[5]

The second type is shown in Figure 6. In this case, the data processor-data memory connection is n-by-n and there is no connection between the data processors. This form of switch is usually called an alignment network. One example of such an architecture is the Burroughs Scientific Processor.[6]

We can now introduce our taxonomy informally. We classify architectures by the number of instruction processors and

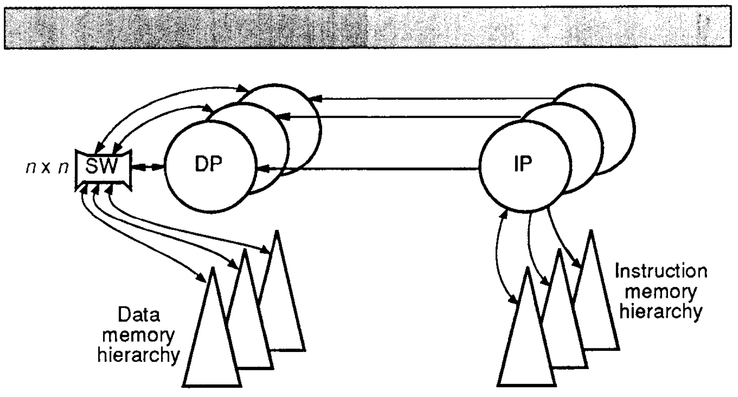

Figure 7. The abstract machine of a typical tightly coupled multiprocessor.

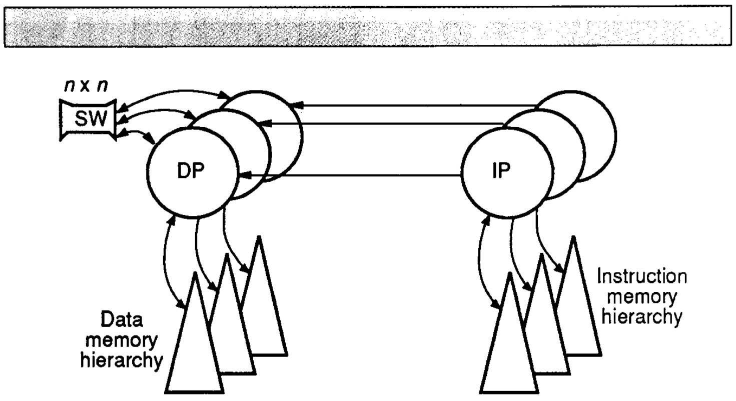

Figure 8. The abstract machine of a typical loosely coupled multiprocessor.

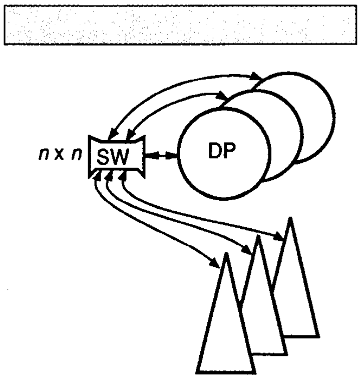

Figure 9. The abstract machine for graph reduction.

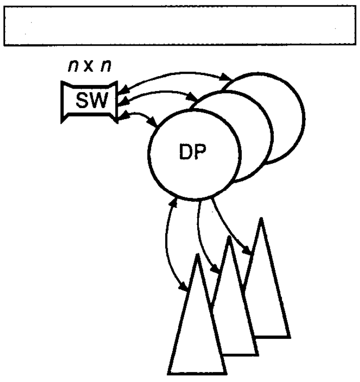

Figure 10. The abstract machine for dataflow (Dennis type).

data processors they have, by the number of memory hierarchies they have, and by the way in which these functional units are interconnected by abstract switches. Thus, a von Neumann uniprocessor can be classified as

- number of instruction processors is 1
- number of instruction memory hierarchies is 1
- switch between instruction processor and instruction memory is 1-to-1
- number of data processors is 1
- number of data memories is 1
- switch between data processor and data memory is 1-to-1
- switch between instruction processor and data processor is 1-to-1

A Type 1 array processor can be classified as

- number of instruction processors is 1
- number of instruction memory hierarchies is 1
- switch between instruction processor and instruction memory is 1-to-1
- number of data processors is *n*
- number of data memories is *n*
- switch between data processors and data memories is *n*-to-*n*
- switch between instruction processor and data processor is 1-to-*n*
- switch between data processors is *n*-by-*n*

We shall see more complex descriptions in the next section. As the number of functional units increases, so do the possibilities for interconnecting them.

Parallel von Neumann machines

Array processors rely on the existence of many pieces of data that can be manipulated in the same way at the same time. Many problems do not fit this paradigm well. Therefore, it is natural to consider replicating the instruction processor as well as the data processor and creating multiple control "threads." This is the approach taken in parallel von Neumann machines. Two very different architecture types result from this approach: tightly coupled systems and loosely coupled systems.

Tightly coupled systems consist of a set of processors connected to a set of memories through a dynamic switch. Any processor can access any location in the memories with about the same latency. Communication and synchronization between processes is achieved by the use of shared variables. Examples of such machines include the BBN Butterfly, the IBM 3081 and 3090, and C.mmp. Loosely coupled systems consist of a set of processors, each with its own local memory. Communication takes place by explicit request from one processor to another over an interconnection network, by message passing, or by remote procedure calls. Examples of such machines include the Intel and NCube hypercube machines, Transputer-based systems such as Supernode (Esprit project 1085) and the Meiko MK40, as well as older systems such as CM*.

In tightly coupled systems, both data and instruction processors are replicated, but the data processors share a common data memory. The corresponding abstract machine is shown in Figure 7. I show it with multiple data memories and an n-by-n switch between data processors and data memories because it is slightly more suggestive of the usual implementation. Functionally, it is indistinguishable from a common shared memory.

Loosely coupled systems also have replicated data and instruction processors, but the switches in the data subsystem differ. A typical loosely coupled system is shown in Figure 8. The connection between data processors and data memories is n-to-n, and there is an n-by-n connection between the data processors.

Machines using other models of computation

Von Neumann machines are all based on a model of computation with threads of instructions executed sequentially, except where the order is explicitly altered. I have already commented that such ordering is over-constrained. This difficulty gets worse when computation models with multiple threads of control are used, since programmers must consider not only the ordering of instructions in a particular thread but the possible orderings of instructions in different interacting threads. It is not surprising that those involved in designing highly parallel machines have examined other models of computation that do not have this awkward property. These new models of computation are characterized by a complete absence of programmer description of the order of evaluation, other than an order implied by data dependencies. This allows any possible evaluation order to be considered for execution, and the one with the greatest performance can be selected by the compiler or the runtime environment. Models of computation with this property are usually expressed in functional languages.

Graph reduction machines. In graph reduction, a computation is encoded as a graph of functions and arguments that (recursively) have the property that the root node is an Apply, the left subtree is the description of a function, and the right subtree is the description of an argument.

The description of a function or an argument can be either a value or a description of how to compute the value. If both the function and the argument have already been evaluated, then the subtree—called a redex—is available for execution. When it has been executed, its value overwrites the subtree. If either has not yet been evaluated, then some subtree must be a redex or the computation cannot proceed. Thus, in general, redexes available for execution are found at the leaves of the tree. Typically, multiple redexes are available for evaluation at any moment, so multiple processors can be involved in evaluation simultaneously.

The abstract machine for graph reduction is shown in Figure 9. It differs from the abstract machines of the von Neumann type primarily in the absence of an instruction unit and instruction memory. The graph being reduced plays the role of both instructions (in its structure) and data (in its content). The data processors and their associated data memories are like those of a tightly coupled multiprocessor, which is not surprising since the graph is a large shared-data structure.

Since there is no instruction processor to provide data labels to the data processor, it must generate them itself. There are two equivalent ways to model this. The first is to provide the data processor with a selector that selects any available redex from the data memory. The other is to regard the data memory as actively pushing the available redexes towards the data processor.

Work on graph reduction has mainly taken place in Europe and the United Kingdom and has been very successful. Several machines have been designed or built. Some examples are Alice[7] and Flagship.[8]

Dataflow machines. Dataflow machines are another class of machine that does not have an instruction sequencing mechanism, other than that implied by data dependencies. Its model of computation is a graph with directed edges, along which data flows, and vertices representing operations that transform the data. Certain edges have a vertex at only one end; these represent the inputs or outputs of the computation. A vertex, or operator, fires according to some firing rule specified by the particular form of dataflow. In general, a firing absorbs a datum from each input arc and produces a result that then departs along the outgoing arc(s).

Most proposed dataflow machines are based on a ring consisting of a matching store (where data values wait until a complete set of operands is present), a memory containing the operators, and a set of processing elements that execute the operators, producing result values that flow around the ring to the matching store.

The abstract machine for a dataflow processor can take two forms, corresponding to the two possible forms of data processor arrangement in an array processor. In the first form, all of the data memory is equally visible to all processors. This approach has been taken in the Manchester Dataflow Machine[9] and in Arvind's machine.[10] This form of processor is functionally identical to parallel graph-reduction machines and, hence, it has the same abstract machine diagram (Figure 9). In the second form, shown in Figure 10, each processor has its own local memory, and data values needed by other processors flow across the inter-data-processor switch. The second form is perhaps more intuitive; in the limit, each processor executes only a single operator. As before, a data processor can be modelled as having a selector or as being sent executable operators by a data memory.

The formal model

Considering the major types of processors that exist today has shown some of the properties that must be captured by a formal classification scheme. My scheme consists of two levels of increasing detail and discrimination. The first level describes an architecture by specifying

- the number of instruction processors,
- the number of instruction memories,
- the type of switch connecting IPs to instruction memories,
- the number of data processors,
- the number of data memories,
- the type of switch connecting DPs

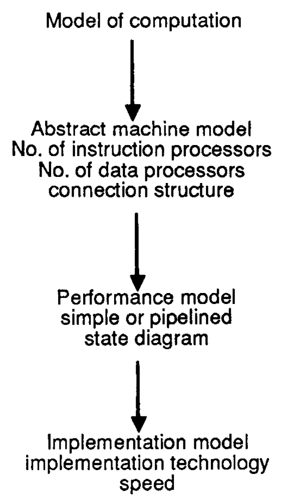

Figure 11. The classification method.

and data memories,

- the type of switch connecting IPs and DPs, and
- the type of switch connecting DPs to DPs.

The first level refines Flynn's classification by expanding the classes of SIMD and MIMD into a set of subclasses that capture important existing architectures.

The second level allows further refinement by describing whether or not the processors can be pipelined and to what degree, and by giving the state diagram behavior of the processors. A third level describing implementation details is also possible. The approach is illustrated in Figure 11.

As we have seen, this classification scheme captures the differences between architectures that are significantly different, while ignoring differences that are primarily a matter of implementation. Table 1 illustrates the set of simple architectures possible under the assumptions that the number of memories matches the number of processors for each subsystem. Although there may be interesting architectures for which these assumptions do not hold, we will ignore them in the interest of a clear exposition.

Classes 1 through 5 are the dataflow/reduction machines that do not have instructions in the usual sense. Class 6 is the von Neumann uniprocessor. Classes 7 through 10 are the array processors. Note that classes 5 and 10 represent extensions to the common architectures of their respective types in that they have a double connectivity for their data processors.

Classes 11 and 12 are the MISD architectures. Although these classes have been treated as an aberration, there are languages for which this type of execution is appropriate. For example, in NIAL (Nested Interactive Array Language),[11] it is possible to write

[f g h] x

that is the parallel application of functions *f*, *g*, and *h* to *x*. I am not aware of any existing architecture of this type, but the con-

Table 1. Possible architectures.

Class	IPs	DPs	IP-DP	IP-IM	DP-DM	DP-DP	Name
1	0	1	none	none	1-1	none	reduct/dataflow uniprocessor
2	0	n	none	none	n-n	none	separate machines
3	0	n	none	none	n-n	n × n	loosely coupled reduct/dataflow
4	0	n	none	none	n × n	none	tightly coupled reduct/dataflow
5	0	n	none	none	n × n	n × n	
6	1	1	1-1	1-1	1-1	none	von Neumann uniprocessor
7	1	n	1-n	1-1	n-n	none	
8	1	n	1-n	1-1	n-n	n × n	Type 1 array processor
9	1	n	1-n	1-1	n × n	none	Type 2 array processor
10	1	n	1-n	1-1	n × n	n × n	
11	n	1	1-n	n-n	1-1	none	
12	n	1	1-n	n × n	1-1	none	
13	n	n	n-n	n-n	n-n	none	separate von Neumann uniprocessors
14	n	n	n-n	n-n	n-n	n × n	loosely coupled von Neumann
15	n	n	n-n	n-n	n × n	none	tightly coupled von Neumann
16	n	n	n-n	n-n	n × n	n × n	
17	n	n	n-n	n × n	n-n	none	
18	n	n	n-n	n × n	n-n	n × n	
19	n	n	n-n	n × n	n × n	none	Denelcor Heterogeneous Element Processor
20	n	n	n-n	n × n	n × n	n × n	
21	n	n	n × n	n-n	n-n	none	
22	n	n	n × n	n-n	n-n	n × n	
23	n	n	n × n	n-n	n × n	none	
24	n	n	n × n	n-n	n × n	n × n	
25	n	n	n × n	n × n	n-n	none	
26	n	n	n × n	n × n	n-n	n × n	
27	n	n	n × n	n × n	n × n	none	
28	n	n	n × n	n × n	n × n	n × n	

cept is not completely ridiculous.

Classes 13 through 28 are the multiprocessors of various kinds. Classes 13 through 20 are more or less conventional multiprocessors with different forms of connection structure. Classes 21 through 28 are more exotic architectures in which the connections between instruction processors and data processors is *n*-by-*n*. These classes appear to be completely unexplored.

Architectures that have an *n*-by-*n* connection between the instruction processors and instruction memories are those in which the decision about the processor to execute a particular instruction is made late (that is, just before execution). Dataflow and graph reduction architectures often have this property, but the only von Neumann machine of which I am aware that behaves this way is the Heterogeneous Element Processor.[12]

Some existing von Neumann machines have more than a single data processor. For example, several high performance pipelined systems (Cray I, Cyber 205) have a data processor for vector instructions and another for scalar instructions. Some have a separate scalar instruction processor as well. Architectures of this type do not fit into the simple scheme outlined above, but they can be easily represented by including the vector processing units with the data processors (number of data processors is $1 + m$).

An example classification for an unusual architecture. I conclude by illustrating the complete description of an unusual processor using my classification. I have chosen the Flagship architecture, a loosely coupled graph reduction system.

The Flagship machine consists of a set of processors, each with its own local memory. The graph to be reduced is partitioned statically, and these partitions are allocated to the local memories of processors. Each processor selects possible parts of its own subgraph for reduction. If the required objects are local, the reduction proceeds and the subgraph is replaced by the reduced value. If some required part of the subgraph is not local (for instance, because it is shared among several different subgraphs), then a request for its value is sent over an interconnection network to the processor in whose local memory the subgraph resides. The value is eventually supplied by the remote processor. A processor that has completely reduced its subgraph can demand more work from other processors.

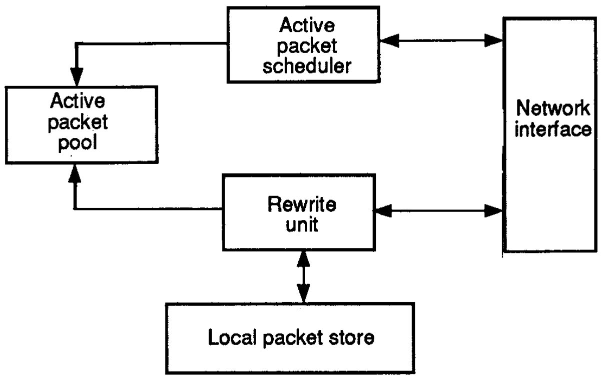

Figure 12. Conventional block diagram of the Flagship architecture.

Graphs are represented as linked structures of nodes called (somewhat inappropriately) packets. A node can be in one of three states: not ready to be reduced (because some subgraph has not yet been reduced), in the process of being reduced, or waiting for some remote object that it needs to continue reduction. This status is recorded within each packet.

The classification of this architecture at the first level is type 3 in Table 1. It corresponds to the architecture shown in Figure 10.

Now, consider the second level classification. The processor architecture in a conventional block diagram is shown in Figure 12. From this conventional description of the processor and its behavior, we can construct a state diagram for the data processor, shown in Figure 13. The way in which the Flagship memory devices form a memory hierarchy is shown in Figure 14. This more detailed description shows that the taxonomy can describe unusual architectures as well as conventional ones, and that such descriptions reveal the internal structure.

I have presented a classification scheme for architectures that is considerably more discriminating than those presently in use. It extends Flynn's popular classification by increasing the discrimination between different kinds of parallel architectures. I have shown that my taxonomy captures the distinctions between architectures where they really significantly differ, and reveals underlying relationships as well (for example, showing the close relation between graph reduction and dataflow architectures). The taxonomy also suggests a number of unexplored possible architectures that might be fruitful places to look for new and innovative machine designs.

The taxonomy is divided into two levels. At the highest level, architectures are distinguished by the number of instruction processors, the number of data processors, the number of memory hierarchies they contain, and by the way in which these devices are connected. At this level, there are 28 simple architectures. At the lower level, further discriminations can be made by describing whether or not each of the processors is pipelined and by giving its internal functional structure by a state diagram. A further level, giving implementation details, is also possible. The emphasis throughout is on capturing the functional behavior of architectures rather than the exact way in which they carry out a set of tasks.

A taxonomy must shed light on what is already known and help in assimilating new understanding, if it is to be useful. I believe that my taxonomy is a natural extension of Flynn's work—more complex because the possibilities have grown, but still simple and straightforward enough to be used as an intellectual tool for understanding and as an engineering tool for design.□

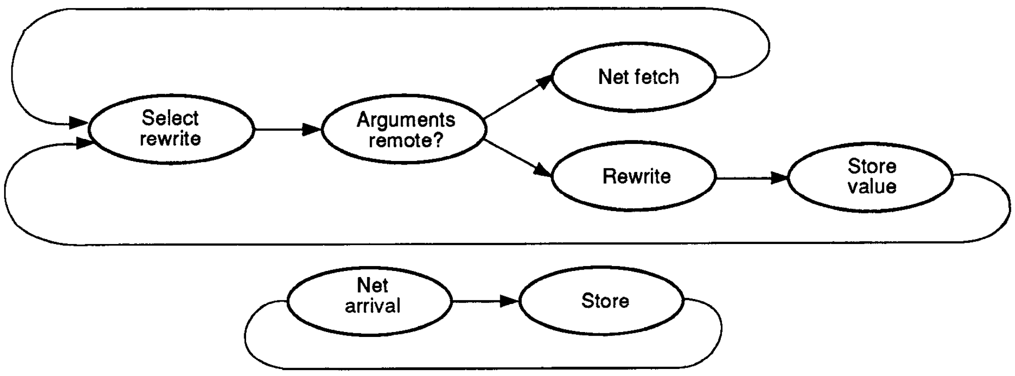

Figure 13. State diagram for the Flagship data processor.

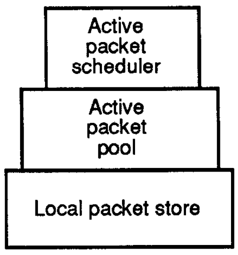

Figure 14. How Flagship memory devices form a memory hierarchy.

Acknowledgments

This work was supported by the Natural Sciences and Engineering Research Council of Canada. David Barnard and the anonymous referees made several suggestions that improved the presentation.

References

1. M.J. Flynn, "Some Computer Organizations and Their Effectiveness," *IEEE Trans. Computers,* C-21, No.9, Sept. 1972, pp. 948-960.
2. T.Y. Feng, "Some Characteristics of Associative/Parallel Processing," *Proc. 1972 Sagamore Computing Conf.*, Aug. 1972, pp. 5-16.
3. S.S. Reddi and E.A. Feurstel, "A Conceptual Framework for Computer Architecture," *Computing Surveys,* Vol. 8, No.2, June 1976, pp. 277-300.
4. W. Händler, "The Impact of Classification Schemes on Computer Architecture," *Proc. Int'l Conf. on Parallel Processing,* Aug. 1977, pp. 7-15.
5. W.D. Hillis, *The Connection Machine,* MIT Press, Cambridge, Mass., 1985.
6. D.J. Kuck and R.A. Stokes, "The Burroughs Scientific Processor," *IEEE Trans. Computers,* Vol. C-31, No. 5, May 1982, pp. 363-376.
7. J. Darlington and M. Reeve, "Alice—A Multiprocessor Reduction Machine for the Parallel Evaluation of Applicative Languages," *Proc. 1981 ACM Conf. Functional Programming Languages and Computer Architecture,* 1981, pp. 65-75.
8. I. Watson et al., "Flagship: A Parallel Architecture for Declarative Programming," *Proc. 15th Ann. Int'l Symp. on Computer Architecture,* May 1988, pp. 124-130.
9. J. Gurd and I. Watson, "Data Driven Systems for High Speed Parallel Computing: Part 1: Structuring Software for Parallel Execution; Part 2: Hardware Design," *Computer Design,* June and July 1980, pp. 91-100, 97-106, respectively.
10. A. and R.S. Nikhil, "Executing a Program on the MIT Tagged Token Dataflow Architecture," *Proc. Parallel Architectures and Languages Europe (PARLE) Conf.*, Springer-Verlag, Berlin, Lecture Notes in Computer Science, June 1987, pp. 1-29.
11. M.A. Jenkins, J.I. Glasgow, and C. McCrosky, "Programming Styles in NIAL," *IEEE Software,* Vol. 3, No. 1, Jan. 1986, pp. 46-55.
12. Denelcor Inc., *Heterogeneous Element Processor: Principles of Operation,* Apr. 1981.

David B. Skillicorn is an associate professor in the Department of Computing and Information Science at Queen's University at Kingston, located in Kingston, Ontario, Canada. He is currently a visiting researcher in the Programming Research Group at the University of Oxford in England. His research interests include multiprocessor architectures and functional languages and how they interact to produce high-performance parallel systems. He is a member of the IEEE Computer Society and the Association for Computing Machinery.

Skillicorn received the BSc degree from the University of Sydney in 1978 and the PhD from the University of Manitoba in 1981.

Readers may write to Skillicorn at the Dept. of Computing and Information Science, Queen's University at Kingston, Kingston, Ontario, Canada K7L 3N6.