## DATA-FLOW AND MULTITHREADED ARCHITECTURES

This article describes the development of data-flow architectures. The execution model of a data-flow computer is radically different from that of a conventional computer in that executability of a data-flow instruction is dependent only on the availability of its input operands. The article starts by discussing the differences between conventional and data-flow architectures. The article goes on to describe a number of representative data-flow and multithreaded architectures that are products of active research done in the area during the late 1970s to early 1990s.

A data-flow architecture is defined as a hardware implementation of a computer system whose computation mechanism is based on a data-driven execution model. Multithreading is a term commonly used by a number of different disciplines of computer science and engineering. In this article, a multithreaded architecture is defined as a hardware implementation of a computer system that is designed to improve upon the computation mechanisms of pure data-flow architectures. In general, a multithreaded architecture is a hybrid whose computation mechanism is borrowed from the data-driven as well as the control-driven execution model.

### Control-Flow Architecture

All conventional computers are based on the ideas proposed by the group from the University of Pennsylvania Moore School that built the ENIAC (1). Some of the key characteristics of a computer from this group are (1) serial execution of instructions and (2) using single memory to store both the instructions and the data. The group's ideas were driven by the need to come up with an architecture that was simple and yet flexible.

The attribute of serial execution gave a simple execution mechanism based on a special register called the program counter (PC) which is updated to point to a memory location that contains the next instruction to be executed. The data that are used as instruction operands are stored at different locations in the same memory. The result produced by executing an instruction is stored back into the memory. The following sequence of instructions represent a program that computes the two roots of a quadratic equation, $ax^2 + bx + c = 0$. The formula to compute the roots is

$$\frac{b \pm \sqrt{b^2 - 4ac}}{2a}$$

```
1.  mult  t1, b, b    ;t1 <= b*b
2.  mult  t2, a, c    ;t2 <= a*c
3.  mult  t2, t2, 4   ;t2 <= 4*a*c
4.  sub   t1, t1, t2  ;t1 <= b*b-4*a*c
5.  sqrt  t1          ;t1 <= sqrt(b*b-4*a*c)
6.  neg   t3, b       ;t3 <= -b
7.  add   r1, t3, t1  ;r1 <= -b+sqrt(b*b-4*a*c)
8.  sub   r2, t3, t1  ;r2 <= -b-sqrt(b*b-4*a*c)
9.  mult  t1, a, 2    ;t1 <= 2*a
10. div   r1, r1, t1  ;Compute r1
11. div   r2, r2, t1  ;Compute r2
```

In conventional computers, instruction execution order is implied by the order in which the instructions appear in a program. In the previous example, instruction 2 is executed after instruction 1 and instruction 3 is executed after instruction 2. Control instructions such as jump and conditional branch allow instructions to be fetched from memory locations that are not located immediately after the current instruction.

In the Moore School group's basic execution model, some instructions may be unnecessarily executed serially. In the previous example, instructions 1 and 2 may be executed concurrently since the two instructions are data independent, that is, one instruction does not need the result of the other instruction to execute. In addition, because the same memory location is used to store different values, artificial data dependencies may occur. In the example, instruction 9 depends on instruction 8. Actually, this is an example of artificial data dependency created by the reuse of variable `t1`. Instruction 9 becomes independent of instruction 8 if a new variable `t4` is used instead.

It is true that in modern computer systems, various hardware and software optimizations are employed to exploit instruction level parallelism (2,3) as just discussed. However, the basic execution model is still that of serial execution where program control is centralized through a program counter.

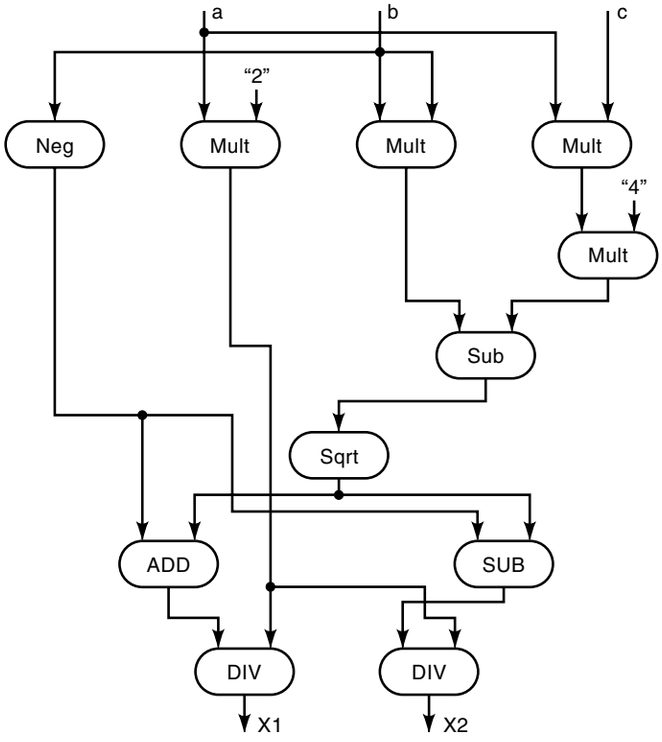


**Figure 1.** The data dependency graph of a program that computes the two roots of a quadratic equation.

## Data-Flow Architectures

Unlike the serial execution model used in a conventional computer, the execution model of a data-flow machine is inherently parallel. In this execution model, the executability of an instruction is determined solely by the availability of the input operands. A program for a data-flow computer is a representation of data dependency graphs where the nodes represent instructions and the edges the data dependency relationship between nodes. The example used in the previous subsection is represented as a data dependency graph in Fig. 1.

We see from the graph that the input operands of the NEG and the three MULT nodes at the top of the graph are the three coefficients a, b, and c. This means that these nodes can execute as soon as the values become available. In the same way, the rest of the nodes are executed as their input operands become available. There is no centralized control mechanism that manages the execution of the instructions. Instead, the execution mechanism is entirely distributed. The data-driven execution model can be viewed as data traveling on the edges of a graph in the form of data tokens. When a node "fires" as a result of its input operands becoming available, the input data tokens are consumed and a new data token is produced which is sent to other nodes that depend on it to execute.

Data-flow architectures are divided into static and dynamic architectures. In a static data-flow architecture, there can only be a single instance of a node at run-time. This means that at a given time, only one data token can travel on an edge of a program graph. On the other hand, a dynamic data-flow architecture allows multiple instances of a node at run-time. This means that multiple data tokens of different instances can travel on a program graph edge. With a dynamic data-flow architecture, it is possible to unfold a loop at run-time, that is, multiple loop iterations can be executed concurrently. With a static data-flow architecture, a loop needs to be parallelized through replication of the loop body nodes by a compiler, for example before the program is executed (4).

In a static data-flow computer, a program graph node can be represented by a template which specifies the operation to be performed, space to store the input operands with associated valid bits, and a list of pointers to the destination graph nodes that need the result of the operation. A data token would need three fields; Data, Port number, and Destination template address fields. The Port number field specifies whether the data is the left or the right input operand. Representation of a node as an instruction template in a memory is shown in Fig. 2.

A static data-flow computer system has three functional units: Update unit, Fetch unit, and Processing unit. The Update unit receives the incoming data tokens and stores the value in the specified instruction templates. If it finds that all input operands are ready, the address of the template is inserted into a ready queue for processing. The Fetch unit removes a template pointer from the head of the ready queue and fetches the instruction template from the template memory and prepares the instruction for execution by the Processing unit. The Processing unit generates data tokens as a result of executing specified operation. Figure 3 shows a schematic diagram of a static data-flow architecture.

A dynamic data-flow computer needs to have a different architecture from that of a static data-flow computer because a program graph node can have multiple instances at run-time. For example, the instruction templates used in the static architecture is not suitable for the dynamic architecture since many data tokens that belong to different instances may be destined to the same node at the same time. Also, a mechanism is needed to detect data tokens that belong to the same instance of a node.

A solution to handle multiple instances of nodes in the dynamic architecture is to assign a unique name to each activation. This is achieved by assigning tags to data tokens. Data tokens belonging to the same activation would have the same tag value. This tag value would be different from the other data tokens that are destined for the same program graph node. The U-interpreter is an abstract data-flow machine using such a scheme (5). By looking at the tag values of the incoming data tokens, it is possible to group those data tokens that belong to the same instance of a given program node.

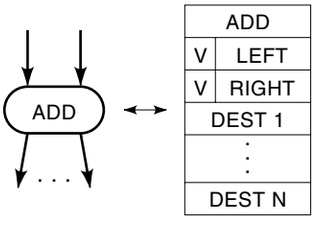


**Figure 2.** Instruction template that represents a program graph node has fields to specify the operation, store input operands and destination node addresses.

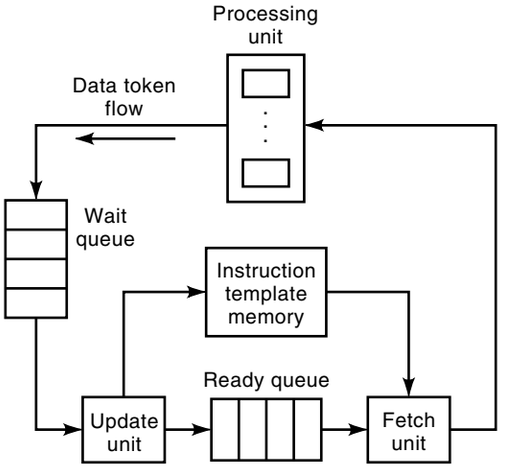


**Figure 3.** Schematic diagram of a static data-flow architecture. Update unit readies instructions according to the arriving data tokens. Processing unit has multiple execution units so that independent instructions can be executed concurrently.

With this approach, a dynamic data-flow architecture needs a special storage where the incoming data tokens are compared to the other tokens already waiting in the storage. If a match is found, the matching tokens are sent to the instruction Fetch Unit. This special memory is usually called the Matching store and is viewed as an associative memory where the tag of an incoming token is compared associatively with all the data tokens waiting in the storage. Naturally, a data token of a monadic node does not need to wait in the memory.

In this section, we have presented a brief overview of the execution model of data-flow architectures and compared it to that of conventional architectures. In the remainder of this article, we discuss the programming languages for the data-driven execution model and present some experimental data-flow and multithreaded computers.

## PROGRAMMING LANGUAGES FOR DATA-DRIVEN EXECUTION

In a data-flow computer, instruction sequencing is determined solely by the availability of the input operands. Therefore, a programming language for a data-flow computer should be such that it is easy to extract the data dependency relationships between operations. This information provides the only constraint to instruction scheduling at run-time. This section presents an overview of Sisal which is an application programming language suitable for data-flow computers (6). In an applicative language, processing occurs by "applying" operators to values which in turn produces other values (4). In Sisal, there are no side-effects as there is no concept of memory. When an operator is applied to operand values, those values are consumed by the operator and result values are produced. This kind of computation model fits well with that of data-flow architectures.

Sisal has program structures that look similar to that of typical modern imperative languages. For example, Sisal has the `if-else` construct which looks similar to that of C. Although syntactically similar, Sisal's `if-else` construct is quite different from that of C semantically. In C, `if-else` is a control statement that determines which statements are to be executed. In Sisal, on the other hand, `if-else` is an expression that returns values depending on the value of the specified condition. The following code segments show the equivalent C and Sisal program statements using the `if-else` construct. The upper code segment is in C and the lower code segment corresponds to the equivalent Sisal code (7).

```
if (i >= j) {
   large = i;
   small = j;
}
else {
   large = j;
   small = i;
}
```

```
large, small := if i >= j then
                   i, j
                else
                   j, i
                end if;
```

In the following example, the `if-else` expression is used inside a larger expression. The `if-else` expression returns either a value corresponding to the sum or the difference of `a` and `b` based on the condition `add`. The returned value is then used in an expression that multiplies it to `c`. Obviously, this kind of a statement is illegal in C. The equivalent C code is shown on the left.

```
if (add)
   ans = a + b;
else
   ans = a - b;
final_ans = ans * c;
```

```
final_ans := if add then
                a + b
             else
                a - b
             end if * c;
```

Sisal has the `let` expression which provides locality of effect. *Locality of effect* means that the data dependencies existing amongst instructions within a given scope are the data dependencies of those instructions within the entire scope of the program. Following is an example of using the `let` expression. In the equivalent C program, the variables `x` and `y` may be global variables in which case it would be difficult to determine the data dependencies among the instructions involved. In the Sisal version, the names `x` and `y` are guaranteed to be visible only within the `let` expression.

```
x = p + 3.7;
y = q + 2.4;
x_times_y = x * y;
```

```
x_times_y := let
                x := p * 3.7;
                y := q + 2.4
             in
                x * y
             end let;
```

Following is an example of one of Sisal's two basic looping constructs. Which loop construct to use depends on whether there are loop carried dependencies or not. If each iteration of the loop is independent, the parallel loop construct is used. Otherwise, the sequential loop construct is used. The one used in the last example is the sequential construct since the indices of the array may repeat, creating loop carried dependencies.

```
for (i = 1; i <= N; i++)
   hist[f[i]] = hist[f[i]] + 1;
```

```
hist := for initial
           temp := array_fill(1,N,0);
           i := 0
        while i <= N repeat
           i := old i + 1;
           temp := old temp[f[i]:
              old temp [f[i]] + 1]
        returns value of temp
        end for;
```

Note that in Sisal, arrays are treated just like scalar values. That means an element of an array cannot be updated. Instead a new array is created based on the old array in which the corresponding array element is an updated value. The keyword `old` is used for such a purpose as shown in the figure. Semantically, the loop in the example produces a new array every iteration. The statement `returns value of temp` returns the array from the last iteration.

An example of using the parallel loop construct is shown next. The loop expression in the upper example returns an array which is bound to `ans`. The lower example is a matrix multiplication. The innermost loop returns a value which is the inner product of two vectors `a[i,*]` and `b[*,j]`. The keyword `sum` specifies the summation reduction operation. Therefore, the innermost loop returns a value bound to `elem` which becomes an element of the resulting matrix `c`.

```
for (i=1; i<=N; i++)
   ans[i] = q + (y[i] * (r *
      z[i+10] + t * z[i+11]));
```

```
ans := for i in 1, N
       returns array of
          q + (y[i] * (r *
          z[i+10] + t * z[i+11]))
       end for;
```

```
for (i=1; i<=N; i++)
   for (j=1; j<=N; j++)
      for (k=1; k<=N; k++)
         c[i][j] += a[i][k]*b[k][j];
```

```
c := for i in N cross j in 1, N
        elem := for k in 1, N
          returns value of sum
             a[i,k] * b[k,j]
          end for
     returns array of elem
     end for;
```

## DATA-FLOW ARCHITECTURES

This section presents three data-flow computers that were built between 1980 and the early part of the 1990s. They are all research machines and have dynamic data-flow architectures. We have chosen these machines due to their significance in the data-flow architecture research. The first of the three was developed by a team from the University of Manchester in the UK (8,9,10). This is the first implementation based on a dynamic data-flow architecture. SIGMA-1 was developed by the Computer Architecture Group from Electrotechnical Laboratory (ETL) in Japan. This is the largest implementation of any data-flow machine built thus far. It consists of 128 processing nodes (11,12). The Monsoon data-flow computer was designed by a team from the Laboratory of Computer Science at Massachusetts Institute of Technology (MIT) in cooperation with Motorola. The significance of Monsoon is in its token matching mechanism (13,14). Although static data-flow architectures are not included in this article, Dennis' work in the development of a static data-flow architecture has stimulated the development of various data-flow projects (15,16).

The common characteristic of these three early data-flow machines is that their actual implementations closely reflect the abstract machine implied by the data-driven execution model. For example, as data tokens are viewed as flowing from one node to another on a program graph, data values are physically transported as data packets. Also, instruction scheduling is done dynamically based on the firing rule defined in the data-driven execution model.

### Manchester Data-Flow Computer

The Manchester data-flow computer has a four-stage asynchronously pipelined structure. The four hardware components are connected via a ring network that is capable of handling up to 10 million packets per second. In addition to the four main components, the computer has an I/O (input/output) switch that connects it to the host. It is also possible to use the I/O switch in a multiprocessor configuration. Figure 4 shows the schematic diagram of the machine.

The Token queue unit is a 32 K Word of hardware circular first in first out (FIFO). A word is 96 bits wide which is the length of a data token. The function of the token queue unit is to store the initial data tokens as well as buffer the incoming data tokens so that the token flow through the pipeline ring is smooth. The data token format is shown here with the

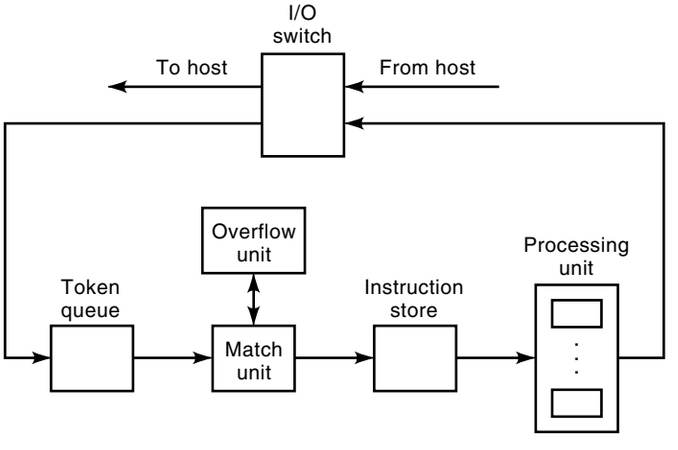


**Figure 4.** The architecture of the Manchester dataflow machine.

field name and the number of bits shown in parenthesis. The `tag` field is used to identify the activation to which the data token belongs. The `marker` bit is used to indicate to which mode (user or system) the token belongs.

```
Token ≡ [data(37).tag(36).destination(22).
         marker(1)].
```

The Matching unit is the module that is responsible for instruction scheduling. An instruction is scheduled dynamically when its input operands become available. The availability of the input operands are determined by comparing the necessary fields of the incoming tokens to each other. Although the matching store can be viewed as an associative memory, the Manchester machine uses a hardware hash table to match tokens due to the high cost of associative memory.

The hash table consists of eight banks of memory that can store 1.25 million data tokens. A 16-bit hashing function is applied to the tag and the instruction address field of the incoming token. The resulting value is used to access the hash table banks in parallel. If a match is found, the matching tokens are formed into a single token and sent to the Instruction Unit. If a match is not found, the incoming token is stored in the matching store. If the hash table space overflows, the token is stored in the Overflow unit.

There are only dyadic and monadic nodes in the program graph which the Manchester machine executes. The Matching unit can match tokens at a rate of 1.11 million matches per second for dyadic nodes and can pass 5.56 million tokens per second for monadic nodes. The Matching unit operates at a clock cycle time of 180 nsec.

The Instruction unit uses the `instruction address` field of the data token sent by the Matching unit. The instruction address field is further divided into the `segment` and the `offset` fields of 6-bits and 12-bits, respectively. The segment and the offset values are used as in virtual memory, that is, the value in the segment field is used to access a 20-bit segment base address from the segment table. The offset is then added to compute the instruction address. An instruction can have up to two destinations. Therefore, a duplicate (dup) operator is needed if there are more than two destinations. Once an instruction is fetched, the instruction and the remaining token fields are formed into a packet and sent to the Processing unit for execution.

The Processing unit can have up to 20 functional units. Each functional unit is implemented using a bit-slice processor. The Processing Unit operates at a clock cycle time of 57 nsec and the fastest instruction takes 16 cycles to execute. The largest number of functional units tested successfully is fourteen. The resulting data is formed into a 96-bit data token which is either circulated back to the token queue unit or out to the host computer.

The Manchester machine used Sisal as the programming language. More detailed information on compilation and performance issues can be found in Refs. 8 and 17.

### Electrotechnical Laboratory SIGMA-1

SIGMA-1 is developed by the Electrotechnical Laboratory in Japan and it is the largest data-flow computer built thus far. It consists of 128 processing elements and 128 structure store elements. Structure store is a special memory subsystem used to store aggregate data types such as arrays. A two-stage Omega network connects the processing and the store elements. There are 32 group nodes in which each node consists of four processing nodes and four structure store elements. Within a node, communication between elements is done via a local network. Figure 5 shows the schematic diagram of the processing and the structure store element.

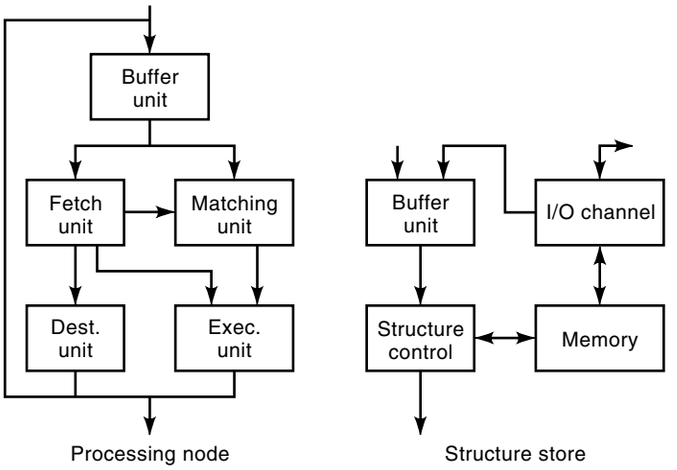


**Figure 5.** Sigma-1 has dedicated hardware structure handlers in addition to the processing elements. The organization of the processing element is shown at left while that of the structure controller is shown at right.

The SIGMA-1 processing element consists of five hardware modules. The function of the buffer unit is similar to the Token Queue unit of the Manchester machine, that is, it buffers incoming data tokens. The Buffer unit can store up to 8 K tokens where a token is 89-bits long. The format of a token is as follows:

```
Token ≡ [pe(8).itag(8).tag(32).c(1).type(8).
         data(32)].
```

The `pe` field specifies the processing node that executes the corresponding instruction. The `itag` field is used to indicate the type of a token, that is, a user or a maintenance. The `tag` field further breaks into four subfields. The format is shown here:

```
Tag ≡ [i(10).base(8).offset(10).flg(4)].
```

The `i` field indicates to which the loop iteration a data token belongs. The loop iteration number is used to identify the instance of a data token when a loop is unfolded dynamically, resulting in multiple iterations that are active concurrently. The `base` and the `offset` fields are used to compute the address of the instruction. The `flg` field specifies the token matching function that should be performed by the Matching unit. One such function is the "sticky" matching function. When the `flg` field of a data token is set to sticky, that token does not disappear after being consumed as an input operand of an instruction. This is useful when the value of a data token is a loop invariant. Without the sticky function, a value that is consumed multiple times must be created as many times as it is consumed.

The three hardware modules, the Buffer unit, the Fetch unit, and the Matching Unit form the first stage of the two

stage pipeline that is driven by a 10 MHz clock. A ready instruction and its input operands are sent to the Destination unit and the Execution unit. The two units operate in parallel. The Destination Unit forms the resulting data tokens as specified in an instruction while the Execution Unit executes the instruction. The Execution Unit has an Integer arithmetic/logic unit (ALU), a floating-point ALU, a multiplier, and a structure address generator.

The programming language used by SIGMA-1 is called DFC (data-flow C) which is a variant of C. It obeys the single assignment rule. More information on the SIGMA-1 can be found in Refs. 11, 12, and 18.

### Massachusetts Institute of Technology Monsoon

One common feature in all dynamic data-flow computers is a hardware module that performs token matching. Ideally, token matching is done using an associative memory that, upon receiving a token, automatically produces the matching token from a pool of tokens stored in the matching store. A matching token is found by associatively matching the tag of the incoming data token with the tags of all the tokens waiting in the matching store. Using associative memory to implement the matching unit, however, is expensive. That is why hardware hash table was used in the Manchester machine. Monsoon is the first dynamic data-flow computer to come up with a mechanism called the Explicit token store (ETS) that provides fast token matching without using associative token matching (14).

In the ETS mechanism, a compiler is used to determine the relative storage location that an incoming data token checks to see whether its matching token is waiting or not. This mechanism is similar to a common technique used in imperative languages in which a compiler assigns local variables of a function on its activation frame. For each function in a program graph, the compiler maps each edge to a location in the corresponding activation frame. An instruction is then generated with the offset of the input operands embedded as part of the code. Since the storage location of every data token for a given activation is known with respect to the base address of its frame memory, associative memory is no longer needed.

The tag field of a data token in Monsoon consists of the base address of its frame memory `FP` and the address of the instruction `IP`. The value of `FP` for a given instance is known at run-time when the corresponding activation frame is created. Finding the matching token of an incoming data token is achieved by following the procedures that follow:

1. The `IP` subfield of the incoming data token is used to fetch the instruction to get the offset value.
2. The offset value is used in conjunction with the `FP` subfield to compute the address in the frame memory.
3. The matching token is waiting in the corresponding frame address if the presence bit is set. In that case, the matching tokens are sent to the execution unit. If the presence bit is not set, then the incoming token is stored and the presence bit is set.

The schematic diagram of the Monsoon processor is shown in Fig. 6. Unlike the other earlier dynamic data-flow machines, the first module that processes an incoming data token is the Fetch unit. This is the direct consequence of the ETS mechanism which needs the offset embedded as part of an instruction to compute the frame memory address that stores the matching data token. When the matching token is found, the tokens are sent to the Functional unit for execution. The tag is computed concurrently. The resulting value and the tag are then formed into a data token and is circulated back into the Fetch Unit or sent out to the network to be processed by a different processor.

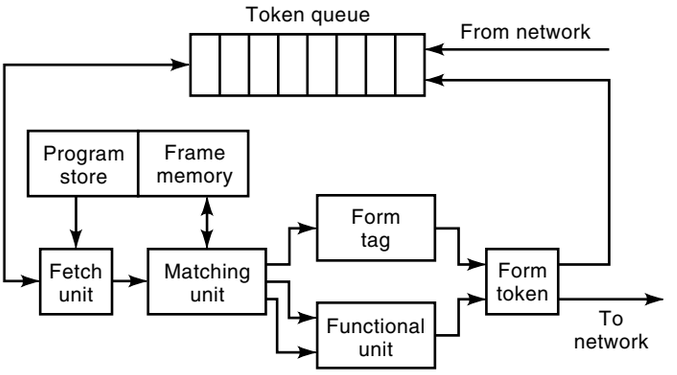


**Figure 6.** Organization of the Monsoon processor.

Monson uses Id as its programming language (19). Id is a functional language developed for data-flow execution.

## MULTITHREADED ARCHITECTURES

Analyzing the performance of data-flow architectures pointed out some drawbacks. The basic approach in all data-flow architectures was the faithful implementation of the data-driven execution model in hardware. This meant providing a token matching capability that checks for the availability of the input operands for every scheduling instruction. This execution model is inherently parallel since the availability of the input operands is the only constraint that determines the executability of an instruction.

One drawback of the data-flow architectures was the expensive hardware required for token matching. The approach taken by the MIT Monsoon is a solution in the right direction. The second problem with the data-flow architectures was that every instruction is scheduled dynamically depending on the status of its input operands. In fact, this is the essence of the data-driven execution model which makes it a parallel execution model. The problem, however, is that even those instructions that are known at compile-time to execute in sequence are scheduled dynamically.

Executing a stream of sequential instructions in a data-flow computer is inefficient, that is, an instruction that needs the result of another instruction cannot start until that instruction completes execution and the result is available. This, however, is not the case in conventional processors. For example, a stream of sequential instructions may be at different stages of execution in a pipeline and the result of an instruction may be sent to the next instruction using techniques such as data forwarding which reduces pipeline stalls (2).

The idea of multithreaded architectures grew out of the realization that it would be beneficial to combine the advantages of the data-flow and conventional architectures. The ap-

proach is that a program code is partitioned either automatically or manually into multiple threads in which a thread is a group of instructions that are executed in sequence. At runtime, threads are scheduled dynamically according to the data-driven execution model. Once a thread is scheduled, however, the instructions inside the thread are executed as in conventional processors. There were a number of proposals for multithreaded architectures (20,21,22).

In this article, we present two multithreaded architectures, EM-4 (22) and P-RISC (parallel-reduced instruction set chip) (20). EM-4 from ETL is the only proposed architecture that was actually built into a prototype. P-RISC evolved from Monsoon and further refined to *T (pronounced "start") (21) for actual implementation.

### EM-4

EM-4 has evolved from SIGMA-1 which had a pure data-flow architecture. EM-4 is a hybrid machine whose architecture has inherited from both the data-flow and conventional architectures. The current implementation of the EM-4 multiprocessor consists of 80 processing nodes connected via a five-stage circular omega network (Fig. 7). Each processing node is based on the EMC-R processor which is driven by a 12.5 MHz clock. The processor has four hardware modules. The Switching unit routes data tokens to itself or to other processing nodes. If a token is destined to the local processor, it is inserted into the Input buffer unit. The Input Buffer Unit has 32 words organized as a FIFO. There is an extra 8 K Word of secondary buffer located in an off-chip memory.

A thread is scheduled for execution if its input operands are available. Once a thread is scheduled for execution, instructions are executed using only the latter two pipeline stages of the processor, the Fetch unit and the Execution unit. Once scheduled, a thread runs to completion if no remote memory read instruction is executed. If a remote memory read instruction is executed, the thread is suspended by the processor and another ready thread is scheduled for execution. The suspended thread is scheduled for execution when the data from the remote memory read becomes available. Long delays caused by remote memory operations in parallel machines can effectively be hidden by performing useful work while a remote memory read is in progress (23).

The programming environment for the EM-4 multiprocessor is an extended C that supports multithreading. A thread is represented as a function and therefore, the granularity of a thread is under the full control of a programmer. A function can be executed as a function through conventional function calling mechanism or as a thread by calling it using a special thread invoking library function.

### P-RISC

The P-RISC architecture is proposed by the same group that developed Monsoon at MIT. The starting point of the architecture, however, was not data-flow. Instead, P-RISC started from conventional sequential architecture and added data-flow features to exploit fine-grain parallelism. In the P-RISC architecture, data do not travel as tokens as long as values are produced and consumed inside the same processor. A thread is completely specified by a descriptor `<FP.IP>` where `FP` is the base address of the activation frame and `IP` is the instruction address.

The two key mechanisms of data-flow architectures are token matching and forwarding of a result as a token to its destination after instruction execution. In data-flow machines, these mechanisms were implemented in hardware and therefore, is transparent to software. In P-RISC architecture, these mechanisms are broken down into primitive operations and made into instructions, hence RISC approach. The result is a simple architecture that is mostly conventional, but with four new additional instructions that enable it to behave like a data-flow processor. The four new instructions are listed here.

1. `fork IPt`
2. `join x`
3. `start v c d`
4. `loadc a x IPr`

The instruction `fork` is used to schedule a thread for execution. When `fork IPt` is executed at address `IP`, a thread starting at address `IPt` is scheduled for execution. A thread is scheduled for execution when its thread descriptor is enqueued to a scheduling queue. As the `fork` instruction sched-

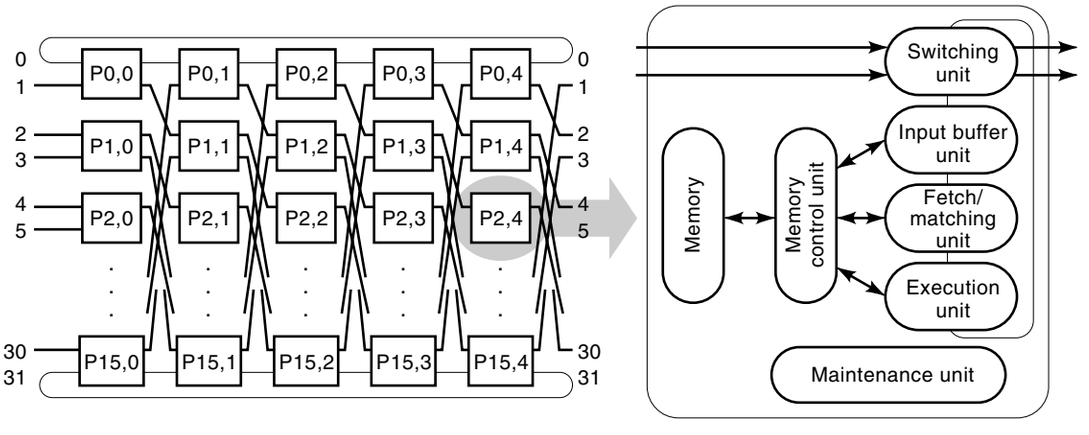


**Figure 7.** The architecture of the EM-4 multiprocessor.

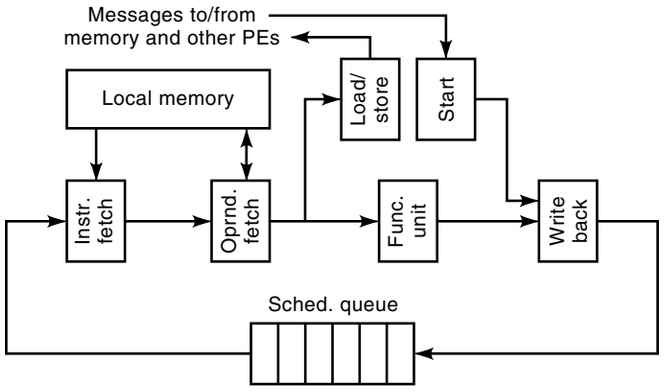


**Figure 8.** The architecture of the P-RISC processor.

ules a thread, the current thread continues execution at address `IP+1`.

The `join` instruction is used to determine whether input operands of an instruction are available. If they are, the corresponding instruction can be executed. When `join x` located at address `IP` is executed, the value stored at address `FP+x` is toggled. If the value was zero, nothing happens. If the value was one, then a thread descriptor `<FP.IP+1>` is issued. The P-RISC processor organization is shown in Fig. 8.

Due to space restriction, the remaining two instructions are not discussed in this article. Interested readers are directed to Ref. 20 for more detailed information. The P-RISC architecture has refined to *T which is based on the Motorola 88110 superscalar processor (24). The idea was to include the message and synchronization unit (MSU) to perform data-flow-like operations (21,25).

## SUMMARY

The data-flow researchers proposed an approach that is a radical departure from the traditional computing which is based on a sequential execution model. The new approach is based on the data-driven execution model which provides a simple and elegant solution to the two main issues of parallel computing, that is, extraction and exploitation of parallelism.

To address the issue of parallelism extraction, functional languages are proposed (26). One major advantage of functional languages is the side-effect free semantics. Such semantics make it relatively easy for a compiler to extract all the available parallelism in a program. As a result, a programmer no longer needs to explicitly specify parallelism to a compiler. Once all the available parallelism is extracted by a compiler, the target architecture must be able to exploit parallelism efficiently. Because the conventional architecture is not efficient at exploiting parallelism, data-flow researchers proposed a new architecture that is based on the data-driven execution model.

Performance of the first data-flow architectures, however, did not meet the original expectations. Some valuable lessons were learned, though. First, the direct mapping of the data-driven execution model to hardware was not competitive enough from the engineering standpoint. For the same amount of hardware, conventional architecture can be designed to yield better performance. Second, while the data-flow architecture was good at exploiting parallelism, it was not very efficient at executing sequential stream of instructions. The lessons learned from the first generation of data-flow architectures led to the development of the multithreaded architectures. The multithreaded architecture is a hybrid that has features from both the data-flow and conventional architectures.

At present, virtually all commercially available parallel and sequential machines are based on processors that have conventional architectures. It is also true that the data-flow ideas form the basis of techniques employed in many of today's high-performance processors that exploit instruction level parallelism (27). Although many people see the computer of the future to be configured as multiprocessors, there is no consensus on its architecture. As the data-flow architecture is evolving from pure data-flow to one that is hybrid, the conventional architecture is also making a similar evolution. As such, the architecture of the future computer would most likely be a hybrid that is efficient in executing sequential as well as parallel code.

## BIBLIOGRAPHY

1. M. Wilkes, *Computing Perspectives,* San Mateo, CA: Morgan Kaufmann, 1995.
2. J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach,* second edition, San Mateo, CA: Morgan Kaufmann, 1995.
3. M. Johnson, *Superscalar Microprocessor Design, Innovative Technology,* Englewood Cliffs, NJ: Prentice Hall, 1991.
4. W. Ackerman, Data flow languages, *Computer,* **15** (2): 15–23, 1982.
5. Arvind and K. Gostelow, The U-interpreter, *IEEE Comput.,* **15** (2): 42–49, 1982.
6. J. McGraw et al., *SISAL Language Reference Manual Version 1.2,* March 1985.
7. C. Kim, J-L. Gaudiot, and W. Proskurowski, Parallel computing with the SISAL applicative language: Programmability and performance issues, *Software—Practice and Experience,* **26** (9): 1025–1051, 1996.
8. J. Gurd, C. Kirkham, and I. Watson, The Manchester prototype dataflow computer, *Commun. ACM,* **28** (1): 34–52, 1985.
9. V. Srini, An architectural comparison of dataflow systems, *Computer,* 68–88, 1986.
10. I. Watson and J. Gurd, A practical data-flow computer, *IEEE Comput.,* **15** (2): 51–57, 1982.
11. K. Hiraki, S. Sekiguchi, and T. Shimada, Status report of SIGMA-1: A data-flow supercomputer. In J-L. Gaudiot and L. Bic, editors, *Advanced Topics in Data-Flow Computing,* Ch. 7, Englewood Cliffs, NJ: Prentice Hall, 1991, pp. 207–223.
12. K. Hiraki, T. Shimada, and K. Nishida, A hardware design of the SIGMA 1- a data flow computer for scientific computations. In *Proc. 1984 Inter. Conf. Parallel Process.,* August 1984, pp. 851–855.
13. D. Culler and G. Papadopoulos, The Explicit Token Store. In J-L. Gaudiot and L. Bic, editors, *J. Parallel Distributed Computing, Special Issue: Data-Flow Process.,* New York: Academic Press, 1990, pp. 289–308.
14. J. Hicks et al., Performance studies of the Monsoon data-flow system, *J. Parallel Distributed Comput.,* **18**: 273–300, 1993.
15. J. Dennis, Dataflow supercomputers, *Computer,* **13** (11): 48–56, 1980.

16. J. Dennis, The evolution of static data-flow architecture. In J-L. Gaudiot and L. Bic, editors, *Advanced Topics in Data-Flow Computing,* Ch. 2, Englewood Cliffs, NJ: Prentice Hall, 1991, 35–91.
17. A. Böhm and J. Sargeant, Code optimization for tagged-token dataflow machines, *IEEE Trans. Comput.,* **38**: 4–14, 1989.
18. T. Shimada et al., Evaluation of a prototype data flow processor of the SIGMA-1 for scientific computations. In *Proc. 13th Annu. Int. Symp. Comput. Architecture,* 226–234, ACM and IEEE, ACM Press, June 1986.
19. R. Nikhil, Id (version 90.1) reference manual. Technical Report CSG Memo 284-2, MIT, July 1991.
20. R. Nikhil and Arvind, Can dataflow subsume von Neumann computing? *Proc. 16th Annu. Int. Symp. Comput. Architecture,* ACM and IEEE, ACM Press, 1989.
21. R. Nikhil, G. Papadopoulos, and Arvind, *T: A multithreaded massively parallel architecture. *Proc. 19th Annu. Int. Symp. Comput. Architecture,* pp. 156–167, Gold Coast, Australia, May 1992, ACM, ACM Press.
22. S. Sakai et al., Design of the dataflow single-chip processor EMC-R, *J. Inf. Process.,* **13** (2): 165–173, 1990.
23. A. Sohn, C. Kim, and M. Sato, Multithreading with the EM-4 distributed-memory multiprocessor. *Proc. 1995 Parallel Architectures Compilation Techniques Conf.,* pp. 27–36, June 1995.
24. K. Diefendorff and M. Allen, Organization of the Motorola 88110 superscalar RISC microprocessor, *IEEE Micro,* **12** (2): 40–63, 1992.
25. M. Beckerle, An overview of the START(*T) computer system, Technical Report MCRC-TR-28, Motorola Inc., Cambridge Research Center, Cambridge, MA, 1992.
26. J. Backus, Can programming be liberated from the von Neumann style? A functional style and its algebra of programs, *Commun. ACM,* **21** (8): 613–641, 1978.
27. R. Tomasulo, An efficient algorithm for exploiting multiple arithmetic units, *IBM J. Res. Develop.,* **11**: 25–33, 1967.

CHINHYUN KIM
Louisiana Tech University
JEAN-LUC GAUDIOT
University of Southern California