# Chapter 4

# Design of a High-Performance Data-Flow Multiprocessor

*David Abramson*
*Gregory Egan*

## 4.1 Introduction

The RMIT/CSIRO Parallel Systems Architecture Project commenced in May 1986. It is a joint collaborative project between the Royal Melbourne Institute of Technology and the Commonwealth Scientific Industrial Research Organisation, Division of Information Technology. The purpose of the project is to investigate parallel algorithms, methodologies, languages and architectures, and in particular architectures based on the data-flow model of parallel computation [1]. The variant of the data-flow model being studied is that first proposed in 1976 by Egan at Manchester University [8] and subsequently further developed at RMIT. A multiprocessor emulation facility is available for high-speed

D. Abramson is with CSIRO, Division of Information Technology, 55 Barry Street, Carlton, Victoria 3053, Australia.

G. Egan is with Swinburne Institute of Technology, School of Electrical Engineering, PO Box 218 Hawthorn, Victoria 3122, Australia.

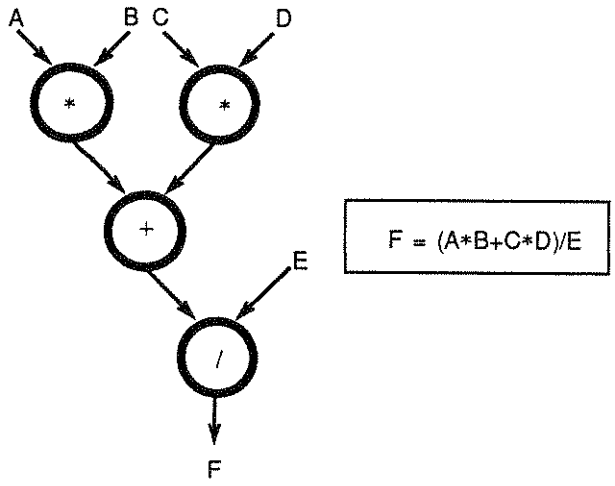



Figure 4.1 A data-flow graph

interpretation of programs as well as a conventional discrete event simulation of the architecture. Compilers for a number of data-flow-like languages are being developed. Work is currently proceeding on the design of processing elements for a high-speed multiprocessor.

This chapter discusses some of the technical design tradeoffs in the construction of the high-speed multiprocessor called CSIRAC II. The target performance for each processor is a sustained diadic node evaluation rate of 5 million instructions per second (MIPS), a sustained monadic rate of 10 MIPS, and a sustained diadic vector rate of 10 MIPS.

A number of graphs are presented to illustrate various features of the machine design. They were taken from a few small SISAL programs, one which calculates $\pi$ by integrating a function [5], and the other which executes a typical numeric loop. Even though these programs are small, their profiles are characteristic of a number of programs which have been run on the RMIT machine simulators, as well as those run at other data-flow sites, such as the MIT Data-flow Project [3].

## 4.2 Data-Flow Multiprocessors

Data-flow machines are multiprocessors which execute parallel program graphs rather than sequential programs. The order of evaluation of the nodes (or instructions) in the graph is determined by the availability of their operands

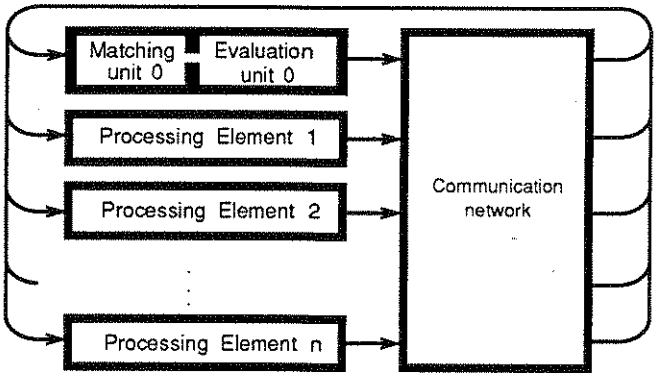



Figure 4.2 Hardware model

rather than the strict sequencing of instructions in a von Neumann machine. Consequently, the program statements are executed in a non-deterministic manner, and concurrency is obtained if more than one node executes at the same time. Figure 4.1 shows a sample data-flow graph for an arithmetic expression, and figure 4.2 shows a model for the hardware required to execute such data-flow programs. In this hardware, the program graph is distributed across the processing elements so that the computation of A*B can proceed at the same time as C*D. The results of a computation are sent from the processor that holds the source node to the processor that holds the destination node. When the result arrives at the destination processor, it waits in the matching unit until all of the operands for the destination node are ready before the result is computed. Thus, the addition is performed when both A*B and C*D have been computed and the division is computed once the addition has completed.

While many simulation studies of data-flow systems have been reported, very few real data-flow processors have been constructed. The RMIT/CSIRO machine has many features in common with other real data-flow processors, such as the Manchester machine, SIGMA-1, Monsoon, and Q-p [10, 15, 12, 13]. The RMIT data-flow machine differs from other data-flow architectures in the following ways:

- The architecture is based on a hybrid of the static evaluation model and the dynamic model.
- The machine supports vector and list operations.
- Graphs are partitioned and allocated statically to processing elements.

- Storage nodes are provided to allow the graph to retain 'semi-permanent' information.
- An Object Store is provided for large structures and persistent objects (e.g., files).

Many of these attributes affect the design of the processors. These effects will be discussed throughout the remainder of this chapter.

## 4.3 The RMIT/CSIRO Data-Flow Machine

The RMIT hybrid evaluation model has been designed to provide efficient implementation of algorithms which require temporal ordering of data sets, as in many DSP applications, as well as highly concurrent scientific codes. The machine enforces data and temporal dependencies by providing token queuing on the arcs of instructions. Data-tokens are consumed from arcs in the order that they are received; thus, they move through a graph in the original order that they were introduced. In this case only one instance of the node need exist, with the data-tokens moving through in a pipelined manner. The machine also allows many concurrent instances of a particular node by tagging the data. In this mode, the many tokens on an arc are consumed in an order dictated by the arrival order of partners on the opposite arc. Concurrency is generated because the tokens can be distributed onto different processing elements even though they pertain to the same instruction.

The effect of the hybrid structure on the machine design is that the matching unit must be capable of supporting both tagged token matching as well as token queueing. The advantages are that code which does not require tagging does not need tagging and untagging instructions, and that the matching store performance is better when it is nearly full. These advantages as well as disadvantages are discussed in [2]. The tags required by the dynamic evaluation model are chosen to be very large so that it is not necessary to re-use them. This removes the need for complex tag management hardware, but does require a larger than normal tag field. This has some impact on the design of the matching unit as well as the evaluation unit.

The RMIT machine supports operations on vectors as well as scalars. The instruction set uses generic instructions which operate on many different data types. For example, if two integers are sent to an add node, then the result is an integer. However, if two floating point numbers are added, the result is a floating point number. In the same way, if two vectors are sent to an add node, then each element of the vectors is added, and the result is a vector. The implementation of vectors as a basic data type allows conventional vector

pipelining techniques to be used for improved performance. The machine must be able to match, build, and transmit vectors as basic types, which affects the matching unit design as well as the evaluation unit.

List operations are supported by using attributes of the hybrid model to keep list structures in order. It is possible to operate on a list, construct a list, and disassemble a list. Start and end of list tokens delimit lists, including lists of lists. It is possible to mix lists and scalar operations on the same instruction. The implementation of lists affects the design of the matching unit because they are supported by special matching functions.

Most instructions in the architecture allow an arbitrary number of output destinations. This affects the design of the evaluation unit because it needs to keep a list of destination addresses, and have them ready in time for token dispatch. In most cases a multiple output node is faster and uses less resources than the more common duplicate trees [14].

The storage node is an instruction which retains a single token which may then be re-read on demand. The data is stored in the matching unit for fast access. This has an impact on the design of the matching unit because it does not necessarily mean that a token is removed from the matching unit when a match occurs. Storage nodes were first described in [9] as a technique for reducing network traffic incurred by the recirculation of loop constants, and also for the construction of fault-tolerant data-flow graphs. They are similar to the sticky tokens used in SIGMA-1, [15]. Large amounts of data are not placed in storage nodes because the matching memory space is a critical resource, and loss of matching space can severely compromise the efficiency of the machine. The object storage mechanism allows storage of large or persistent data structures and is implemented by the evaluation unit. Objects can either be direct read write structures with no automatic synchronization between reads and writes, or can be I-structures [4]. I-structures provide read before write synchronization, and prohibit write after write operations, to guard against indeterminate results.

The code partitioning scheme used by the machine distributes work around the multiprocessor giving excellent work load distribution. Two work distribution schemes are used in combination. The first statically places the nodes of a graph by randomly choosing a processor number at compile time. In this way, the graph is randomly distributed across the multiprocessor. If this code is executed with tokens of the same color, or tokens with no color, then the destination address for a node is the one computed at compile time. However, if the tokens are colored, then the color of the incoming token is hashed with the statically computed destination processor number to form a dynamic processor number. This scheme has the effect of distributing the work for multiple instances of a graph around the multiprocessor. In the latter case, the code is physically placed on all processors. These distribution schemes have the advan-

tage that they are relatively easy to implement, and are very fast. In order to determine the effectiveness of the work load distribution, a number of programs were executed on the data-flow simulator. A base level critical path length was computed for each graph at run time, and this was compared with the effective path generated by the randomization schemes. In most cases, the randomization performed within 70% of the critical path [14]. A disadvantage of this work load distribution scheme is that there is very little locality as most tokens are required on a different processor from the one which generated them, which has a large impact on the design of the network and matching store. Later we discuss a network capable of meeting the high bandwidth required for the target evaluation rates.

In the following sections, we will discuss the details of the processing elements with emphasis on the processes associated with operand matching.

## 4.4 Processor Structure

Figure 4.3 shows the overall structure of a processing element. The element is split into two main functional units. The matching unit accepts tokens as they arrive from the network, and builds work packets for evaluation. Each work packet consists of a function name, a process number, a node number, a color (or tag), and up to two operands and their types. The evaluation unit applies the function to the operands and sends the results through the network to their destinations.

The matching unit has a large input queue for holding tokens which have arrived from the network, but have not been matched or stored. This queue needs to be sufficiently large to hold excess tokens generated while a computation is growing.

Tokens are held in a matching memory while they await their partner. This memory behaves as a large associative store, retrieving tokens based on their node numbers (22 bits) and their colors (40 bits). Because the key field is large, a fully associative memory is impractical, and a token cache coupled with a secondary hash table in bulk memory is used.

To account for variation in the matching-class and the evaluation-function times, from token to token and function to function, a small queue of work packets is maintained between the units. In the evaluation unit, simple functions are handled directly in one machine cycle, with more complex operations being supported by microcode. In the normal case, the function from the evaluation queue is applied to the ALUs, and the operands are processed directly. The results are passed to the dispatch section of the evaluation unit for transmission over the network.

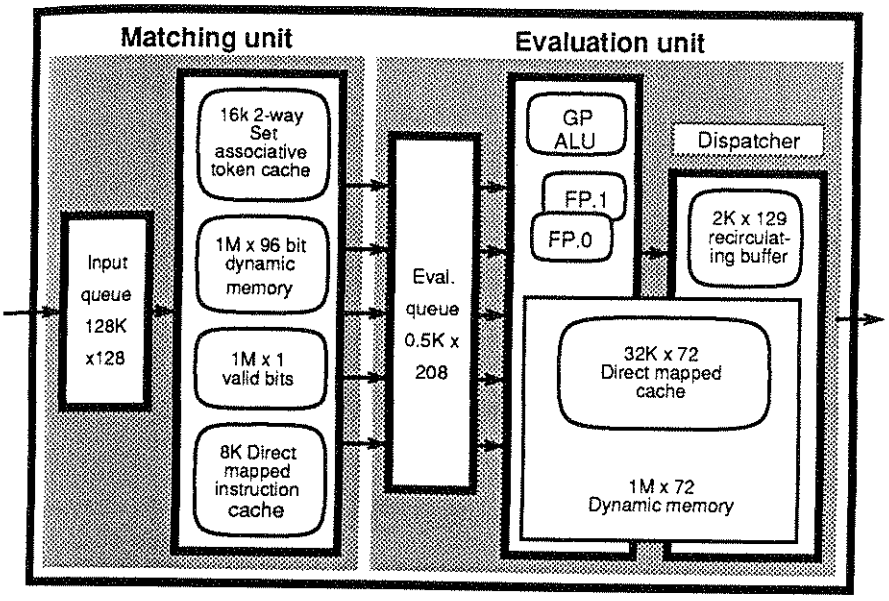



Figure 4.3 Processing element structure

### 4.4.1 Matching unit

The matching unit is responsible for detecting when both operands of a diadic node have arrived, or when one operand of a monadic node has arrived, and building a work packet for the evaluation unit. The matching unit in the RMIT machine has to perform match operations on both tagged (or colored) tokens as well as uncolored tokens. When a tagged-token arrives, the matching unit searches for another token with the same tag and node number. If one is present, and it is for the opposite input point, then it is consumed. If one is present but for the same input point, then the new token is added to the end of a queue. When a token is removed because of a match, it is always removed from the head of the queue. When a token arrives without a color it behaves as though the color was zero.

The matching unit also handles storage nodes and list operations. In the case of a storage node, the data may not be removed from the matching store when a match occurs and can be read out repeatedly until the node is reset or the data overwritten. List operations are qualified by detecting start and end of list tokens.

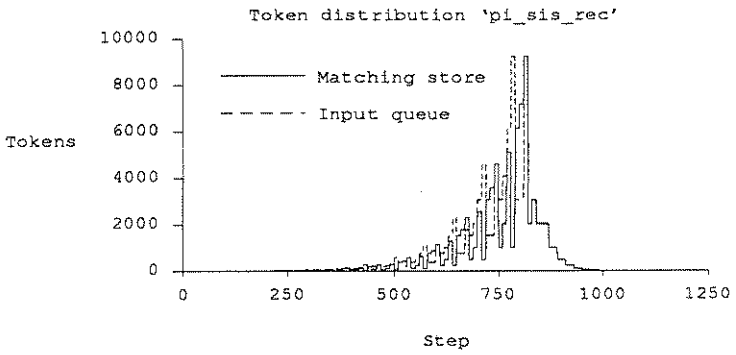



Figure 4.4 Token storage requirements for the calculation of $\pi$

### Input queue design

The input queue poses a design problem because it needs to be both large and fast. It must be capable of meeting the peak transfer rate of the network, but must also be able to hold most of the tokens generated during the buildup phase of a computation. Commercial queue and queue management chips are typically quite small and do not support large data sets.

Figure 4.4 shows the characteristic form of the storage requirements of a typical data-flow program. It plots the number of tokens present in the matching unit as well as the input queue against time. It illustrates that the input queue needs to be roughly the same size as the matching unit in order to handle the buildup of tokens while the computation is growing. It is important for the matching unit to be as large as possible, and consequently the input queue needs to be large. The matching unit has been designed to hold 256k tokens and it was convenient to set the input queue size to 128k tokens.

The input queue has been designed to meet a peak network, and processor rate of inserting and removing one token every 50 nSec (for discussion of this rate see section 4.4.3). It is constructed using a four way interleaved memory structure from 100 nSec memory devices as shown in figure 4.5.

Tokens are composed of one or more 128-bit words. The format of a single word token is shown in figure 4.6. The first word of a token contains a monadic match flag, processor number, process number, node number, input point, color, type, and 40-bit data field. The monadic match flag indicates whether a match is required, or whether the token can be passed directly to the evaluation queue; the process number is used to distinguish different users, or tasks, in the machine; the input point indicates which input of the node the

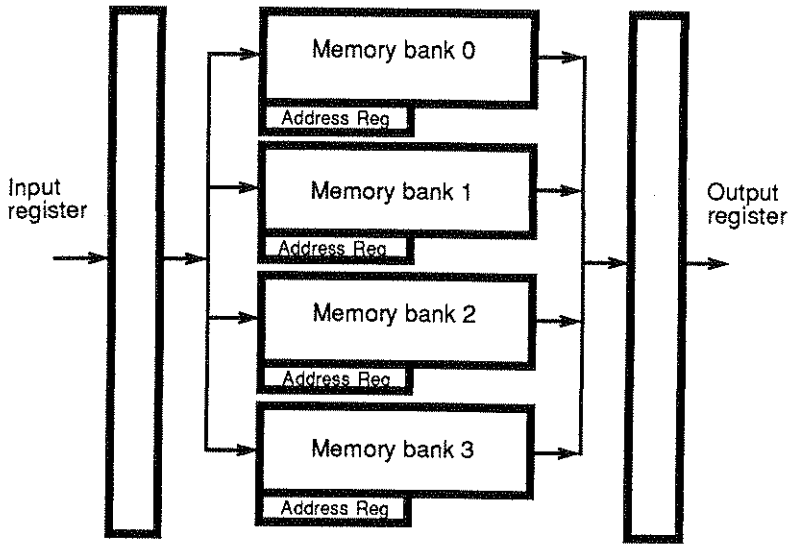



Figure 4.5 Interleaved input queue

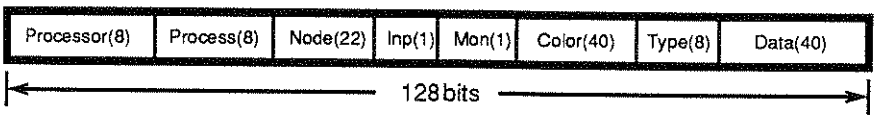



Figure 4.6 A token

token is directed to; and the type field indicates the type of the data. The data field is sufficient to hold the most common data types.

If a token consists of multiple words, as in the case of vectors or compound structures, then subsequent words hold the data fields. In the case of a vector, each word following the first word would, in the case of 32-bit reals, contain four elements of the vector. Thus, a four-element vector only requires two words. Records are packed into two or more token words.

### Matching process

When a token arrives, a high-speed associative search must be made to determine whether a partner is available. The RMIT machine holds waiting tokens in a hash table. The node and color fields are hashed into a primary table, as shown in figure 4.7. All synonyms are stored in a linked list. When a token arrives the entire chain is searched, and if no matching token is found, it is

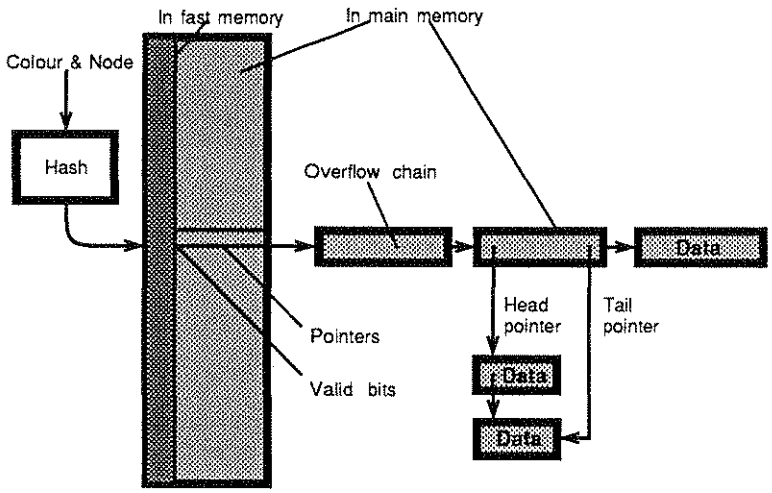



Figure 4.7 Hash table structure

placed at the head of the synonym chain. If a queue is not present, then the entry holds the data directly. If a queue is present, then it contains head and tail pointers to the queue.

The main hash table is designed to be sparsely occupied, so that the probability of an overflow chain is very small. In the RMIT machine, this table is large enough to hold four times the maximum number of tokens; thus, the chance of an overflow chain is 0.125 [11].

When a tag is not present, the hash table becomes a direct access table, indexed by node number. In this case the overflow chain would only contain one entry, although it is still necessary to verify the node number in the case where a graph contains both colored and uncolored tokens. The hashing function combines the node number with the color field using a multi-way exclusive-or function. When the color is not present, only the node number is used.

In order to speed access to this matching table, a special token cache holds the most recently written tokens. Thus, the hash table is only accessed when the token cache misses and the partner has already been stored in the hash table.

### Token lifetimes and caching

Conventional processors make use of locality to speed up memory accesses by storing the most recently used data items in a high-speed cache memory. Such caches work because once a location has been accessed, it is a likely candidate

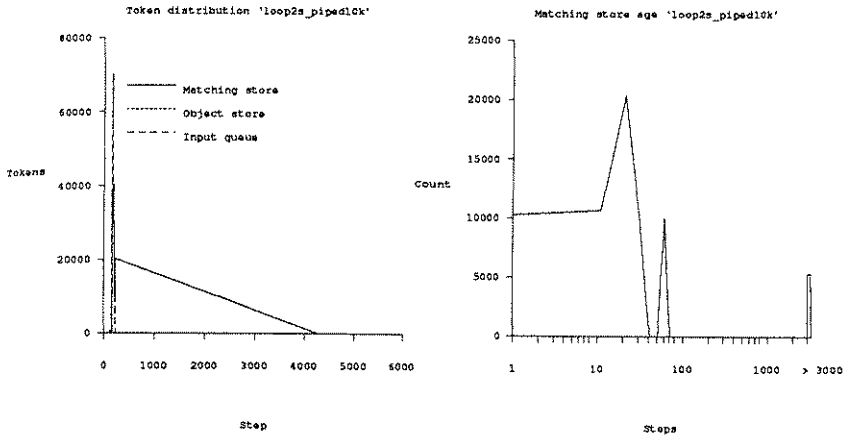



Figure 4.8 (a) Storage requirements for loop; and (b) Token ages for loop

for future references. Typical cache hit rates can be very high. However, the matching process in a data-flow machine is such that a token either matches with its partner, or is stored and awaits a partner. Once retrieved, a token is normally discarded from the matching unit. Thus, the locality exhibited in conventional machines is not present in data-flow machines.

However, recent research has indicated that there is actually a significant amount of temporal locality in the arrival times of tokens in a data-flow machine. Once a token is stored in the matching unit, its partner will usually arrive shortly after the original token. Figure 4.8(a) shows the token storage requirements of the loop program and figure 4.8(b) shows the token lifetimes of the same program. It can be seen that the average age of most tokens is very short (about 20 steps) even though the number of tokens generated by the program is quite high (an average of 10000 tokens for 4000 steps). Thus, it is possible to place tokens in high-speed memory when they arrive in the expectation that their partners will arrive within a short time interval; this approach was first proposed by Brobst in [7]. When the token cache becomes full, older tokens are retired to some other main memory based associative store. A difficulty with this scheme is that only half of the incoming tokens will match, thus only half of the searches can be performed in the fast cache. In the other cases, the secondary memory must still be searched to determine whether the token's partner is already present. Brobst proposed a technique where tokens are always placed in the cache if their partner is not in the cache even though it may have retired to main memory. However, when they are ejected from

the cache into main memory, a search could take place. This scheme allows matching tokens to remain unmatched even though both are present, and could lead to difficult situations including processor starvation and deadlock.

The RMIT machine uses a cache to hold recently unmatched tokens, but also uses a set of valid bits in high-speed memory to indicate whether the partner may be in main memory. The valid bits are taken from the main hash table structure, and indicate whether there are any tokens in the particular overflow chain of the main hash table. If the valid bit is set, then the token's partner may be present, and a search is required before the token can be inserted in the cache. By making the hash table large enough, the probability of false triggers can be made acceptably low.

### Instruction cache

The evaluation patterns of data-flow programs are similar to conventional von Neumann programs, and program loops exhibit significant temporal and spatial locality. The RMIT machine uses a separate direct mapped instruction cache for holding the most recently used instructions. It contains the function code as well as any literal operands attached to the instruction. The cache is read only, and is loaded from main memory when an instruction is not found. If a literal is present, then it is loaded into the cache at the same time as the instruction.

### Matching of queued, record, and vector tokens

The token cache is implemented as a conventional two-way set associative cache memory, with a fixed size data field of 48 bits. This will allow 32-bit data and 40-bit addresses to be stored directly in the cache in the case when there is no queue present on an input. When a queue is formed, the entry is used to hold head and tail pointers to the queue (24 bits each), which is then placed in secondary memory. While the formation of a queue actually slows down the matching of those tokens which are queued, it does not affect the access time of tokens not in the queue. This is because the queue only consumes one cache location, or one hash table location, regardless of the size of the queue. In a machine which does not support queues, the tokens must all be stored directly in the cache for immediate access, even though they will be processed in FIFO order. This has the effect of increasing the cache or hash table occupancy, which causes a decrease in match rates. Further, if queueing is required, then code must be placed in the graph to enforce queueing. These issues are discussed further in [2, 14]. In most cases, a queue will not be present, even in static code, so the cache can process operands at the peak rate of 10 million match/mismatches per second. Thus, the provision of queueing at the architectural level does not affect the machine performance in the likely case

that no queues actually form. Further, the hardware does not seem to have been complicated by allowing queues to be enforced, because microcode is used to perform matches when a queue forms on a node input. Thus, the only complication is that extra microcode is required over the normal code required to handle a cache miss.

The architecture also supports records and vectors as basic tokens. These tokens are of varying sizes, and thus cannot be stored directly in the token cache. When a record or vector token is received, an entry for the token is written in the cache, but the data area holds a pointer to the token contents, which are then stored directly in main memory. Each word of the structure is stored in contiguous space, and can be read out sequentially, faster than normal random accesses to the memory, by using page or nibble mode dynamic memories. Thus, even though random access to the main memory is slower than the cache memory, the information can be saved and retrieved quickly. In the case of vectors, once the start-up cost has been incurred, it is possible to retrieve two elements of the vector every 100 nSec, thus giving a peak vector transfer rate of 20 million elements per second. This will then yield a sustained diadic vector evaluation rate of 10 MIPS.

Direct support for vectors allows a much faster execution rate than could otherwise be achieved. First, four elements of the vector are packed into the 128-bit token words, reducing the network traffic significantly. Second, the vectors are stored in slower dynamic memory instead of occupying prime locations in the token cache, improving cache performance. Third, vector elements are transferred to the execution unit two elements per work packet, which doubles the normal execution rate. At the time of writing this chapter, the vector start-up cost had not been accurately determined; however, it is expected that an overhead of about five microinstructions is necessary before vector elements are transferred at peak rate. Ignoring the first two performance effects, this would yield a vector start-up cost of ten elements. It is expected that in a multiprocessor data-flow machine, vector operations would be processed by both using the vector representation as well as loop unraveling. Thus, each processor could process a section of the vector using the vector representation.

### Controlling the match process

There is a significant difference in the control sequences required when a token is placed in the cache, and when a token must be inserted in main memory. The matching unit control system is optimized so that the cache control is performed by a high-speed state machine, but the more complex operations involving main memory are controlled by a microprogrammed control unit. The microprogrammed control unit normally spins in a microcode loop, issuing

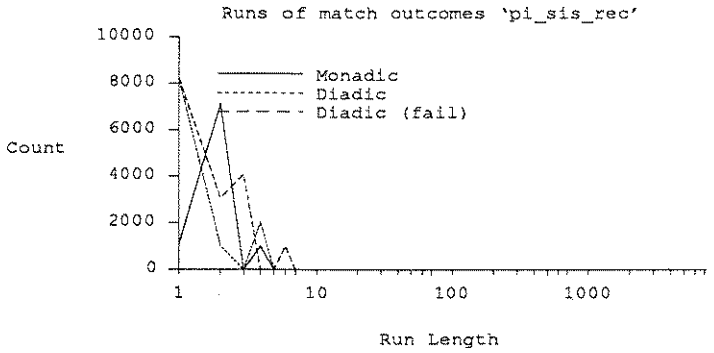



Figure 4.9 Runs of monadic/diadic tokens

an instruction which allows the cache control unit to match or mismatch simple tokens. However, when an exception occurs, the microprogrammed control unit exits the loop and handles the situation directly.

This technique has the advantage that the most common match operations which can be handled in one machine cycle execute at the maximum rate, and the more complex and less common cases can be dealt with by more general, albeit slower, microcode. The microprogrammed control unit occupies less than 10% of the space required for the matching unit.

### 4.4.2 Evaluation Unit

The evaluation unit supports computational, organizational, and structure accessing functions. The major elements of the unit are discussed in the following sections.

#### Evaluation queue

Figure 4.9 shows that a typical data-flow program can generate runs of tokens which can affect the performance of the machine. It shows the frequency of monadic only runs and diadic only runs in which the token's partner is either present or absent. The effect of a run of tokens for a monadic node, or tokens for a diadic node in which the partner is already present, is that a work packet is emitted every cycle. This means that the evaluation unit may receive an excessive amount of work, because not all functions can be executed in a single cycle. Conversely, a run of tokens for a diadic node in which the partner is not present means that the matching unit will not emit a token, leaving the

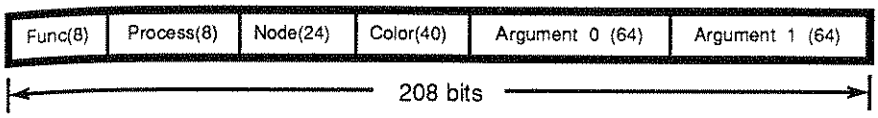



Figure 4.10 Evaluation queue entry

execution unit starved of work. Further, not all match functions take the same time to execute, which can also cause starvation in the execution unit.

A queue is placed between the matching unit and the evaluation unit to compensate for bursts of monadic or diadic matches and for data-dependent variations in operation times of the matching unit and the evaluation unit. If a token is destined for a monadic node, then the matching unit can forward work packets at the rate of 10 million per second. However, for normal diadic nodes, the rate falls to 5 million per second or less for more complex match classes and complex data types. The operation times of the evaluation unit vary depending on the complexity of the function to be performed, the data types of the operands, whether a stored object access is required, and the number of result destinations.

Each entry in the evaluation queue is a complete work packet. It contains the function name, process number (for multiprocessing), node number, color, operand types, and operand data. The format of a queue entry is shown in figure 4.10. The evaluation queue is small relative to the input queue, and is constructed from commercially available 512-word deep queue devices.

### Function evaluation

Function evaluation is supported by a general purpose ALU and two floating point/integer/logical ALUs. The output of the evaluation queue is sent to all ALUs for processing as shown in figure 4.11.

The evaluation unit executes most functions using a data driven control scheme whereby the function and argument data types (e.g., integer, real, boolean) are applied directly to the functional units without employing the microcode control word. This allows the most frequent functions to be executed in 100 nSec. In more complex functions, or where type coercion is required, the instruction may require more than one cycle, in which case a microcode sequence is started to manage function evaluation. In this case, the type and function fields are used as an index into the microcode so that the correct sequence can be started without an expensive sequential decoding process.

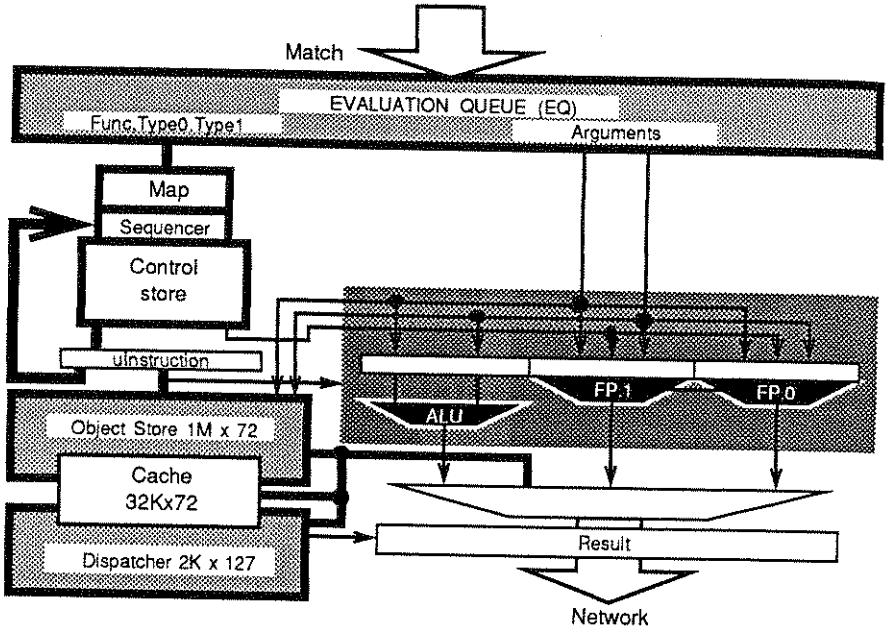



Figure 4.11 Evaluation unit

### Token dispatch

Each node description has a list of destination addresses which are stored as quads in two contiguous 72-bit main memory locations. An example of a three entry destination list is shown in figure 4.12. Each half of a memory location has a 2-bit field indicating its disposition. The instruction node number is scaled by two to provide a start address for the destination list.

If more than four destinations are required, then the fourth destination field, as indicated by its disposition tag, is used as an indirection pointer to unused destination fields in other quads. In the normal case, two destinations are fetched while a function is being evaluated. When a token is ready for dispatch, the first destination address is merged with the result data before it is written to the network. Because the second destination is already available, a result may be redispatched in the next 50 nSec cycle. Subsequent destination pairs are either prefetched every 100 nSec, using page mode from dynamic memories, or are retrieved directly from a cache of destination list entries. An indirection causes a missed 50 nSec cycle. As the ALUs operate concurrently with the dispatch process, multi-word results are copied into a recirculating

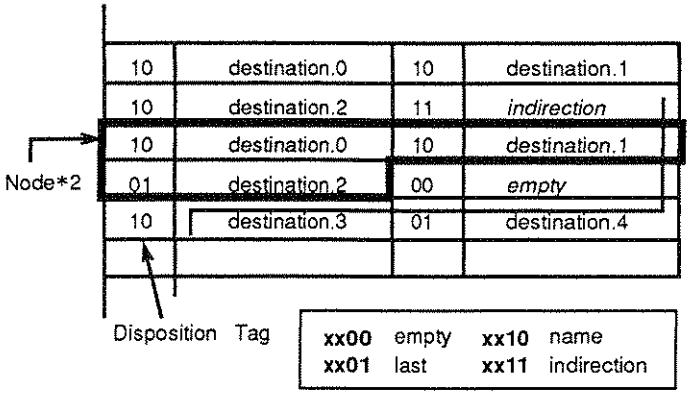



Figure 4.12 Destination lists

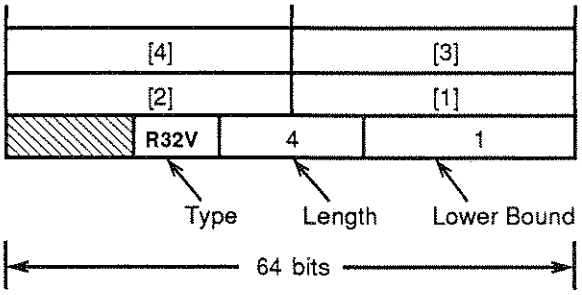



Figure 4.13 Vector arguments

buffer for redispatch.

### Vector function evaluation

Vectors are passed to the evaluation unit in the format shown in figure 4.13. The first word of the work packet contains the length and lower bound of the vector, and subsequent words contain one or more elements of the two operands. If the vector data type can be contained in 32 bits, then each work packet contains two elements of each vector. In this way, the two floating point processors in the evaluation unit can operate on the vector elements concurrently. The sustained vector rate will therefore be 10 MFLOPS, compared with the 5 MIPS scalar rate.

#### Object store

The execution unit is responsible for managing the object store. It can create, destroy, and access objects. An object may reside either entirely within one processing element, or can be distributed across the multiprocessor. The latter form, which would most commonly be used for vector and array structures, reduces structure contention but increases latency. A more detailed description of object store operations will be the subject of a later document.

### 4.4.3 Communication network

The data-flow multiprocessor is composed of a number of identical processing elements connected by a high-speed interconnection network. As was described in section 4.3, work load is distributed by two randomization processes. One allocates the instructions statically at compile time, and the other distributes the token traffic randomly at run time. The combination of the two yields excellent work load distribution [14]. However, the disadvantage is that all tokens generated must be transmitted over the network. A major feature of data-flow machines is that they can tolerate moderately high levels of latency in the network because a processor does not remain blocked while it waits for data [3]. However, the tokens must still arrive at a rate which allows the processor to achieve its peak instruction rate. For example, a 5 MIPS processor must receive two tokens for diadic nodes every 200 nSec, or one token every 100 nSec. If the nodes are monadic, then only half this rate is required. Thus, the network has to have a very high bandwidth, but may be many stages deep and thus have a significant latency without unduly affecting performance.

The RMIT machine implements its communication network with a multi-stage interconnection network (MIN). These networks are typically composed of crossbar switches and have at least one pathway between any two ports. The properties of MINs have been studied extensively in the literature, and a recent review can be found in [6]. The design and construction of a MIN used in a multiprocessor interpreter of the RMIT architecture is described in [16]. The most important design parameter for this class of network is the probability of acceptance of a token, Pa. This parameter indicates how likely it is that a token will be accepted for transmission, and it affects the average bandwidth of the network. In general, the larger the network, the lower the value of Pa. The consequence of a decreasing Pa is that the network must operate at a faster peak speed in order to meet a given average rate. For example, if Pa is 0.5, the network must have a peak speed of twice its required average speed. Figure 4.14 shows the expected values of Pa for a varying size MINs constructed from 2x2 and 4x4 crossbar switches.

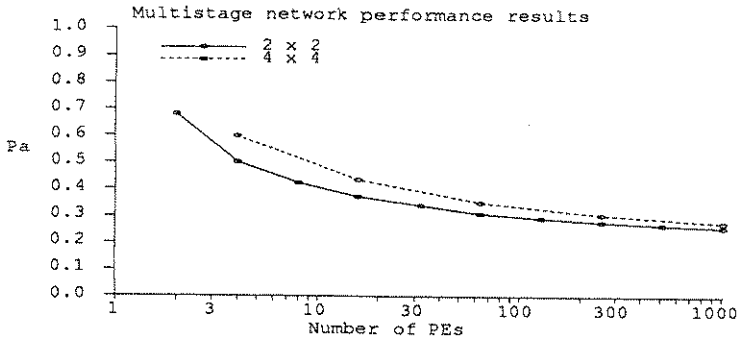



Figure 4.14 Pa for various MIN sizes

### Switch design

The RMIT machine will use a buffered synchronous MIN built from 4x4 crossbar switches. The switches operate on a basic cycle time of one transfer every 50 nSec, which is twice the rate actually required to maintain the processor performance. The reason for this is that the probability of acceptance falls to about 0.45 for a two-level network of 4x4 switches. Thus, to achieve a transfer every 100 nSec, the switches must actually operate at 45 nSec intervals, which for convenience has been rounded up to 50 nSecs. The same rate applies to the input queue and the output stage. A two-level network would support 16 processors, providing a sustained vector performance of 160 MFLOPS. Increasing the size of the machine above 16 processors would require a faster network in accordance with the lower expected Pa value, as well as increased speed of the input queue and token dispatch unit. The remainder of the processor would *not* require alteration.

The prototype network will be constructed from conventional PAL devices, and is expected to achieve the target speed easily. Later versions of the switches will be implemented either from gate array technology or commercially available switches, should they be available. An important observation is that the network can be constructed from the same speed devices as those used in the processing elements. In this way, the entire system can be scaled as faster logic becomes available.

## 4.5 Conclusion

In this chapter, we have discussed some of the important issues that need to be considered in the design of a high-speed data-flow multiprocessor. We have illustrated the sections which have the most effect on performance, and have shown that it is possible to construct these from current, conservative technology, devices. With faster logic families and/or semi-custom silicon implementations of the critical sections of the processor, a much faster processing element could be constructed.

Current work includes the construction of a 4-element multiprocessor, and the development of a suitable multistage interconnection network. Future work will be involved in measurement of system performance, and considerations involved in mapping critical sections of the processors onto silicon.

### Acknowledgments

The authors wish to acknowledge the support of the project team, in particular Mark Rawling, Neil Webb, Paul Whiting and Allan Young. Special thanks go to Mark Rawling, who reviewed earlier drafts of this chapter and formatted the final copy, and to Allan Young who produced the statistics for the multistage networks. Also, special thanks go to Stephen Brobst from the MIT Computation Structures Group for his helpful comments. The Parallel Systems Architecture Project is a joint project between the Commonwealth Scientific and Industrial Research Organization (CSIRO) and the Royal Melbourne Institute of Technology (RMIT).

## References

[1] D. Abramson and G.K. Egan. An overview of the RMIT/CSIRO parallel systems architecture project. *Australian Computer Journal*, August 1988.

[2] D. Abramson and G.K. Egan. The RMIT data flow computer: A hybrid architecture. *The Computer Journal*, June 1990.

[3] Arvind, S. Brobst, and G. Maa. Evaluation of the MIT tagged-token dataflow architecture. Technical report, Laboratory for Computer Science, MIT, November 1987.

[4] Arvind and R.E. Thomas. I-Structures: An efficient data structure for functional languages. Technical Report MIT/LCS/TM-178, Laboratory for Computer Science, MIT, September 1981.

[5] R. Babb. *Programming Parallel Processors*. Addison-Wesley, 1988.

[6] L. Bhuyan, Q. Yang, and D. Agrawal. Performance of multiprocessor interconnection networks. *IEEE Computer*, 1989.

[7] S. Brobst. Instruction scheduling and token storage requirements in a dataflow supercomputer. Technical Report CSG-Memo-264, MIT, June 1986.

[8] G.K. Egan. *Data-flow: Its Application to Decentralised Control.* PhD thesis, Department of Computer Science, University of Manchester, 1979.

[9] G.K. Egan. A decentralised computing system based on data-flow. In *IEEE Industrial Control and Instrumentation Conference*, March 1980.

[10] J. Gurd, C.C. Kirkham, and I. Watson. The Manchester prototype dataflow computer. *Comm. A.C.M.*, 28(1):34–52, January 1985.

[11] R. Morris. Scatter storage techniques. *Comm. A.C.M.*, pages 38–43, January 1968.

[12] H. Nishikawa and H. Terada. Architecture of a one-chip data driven processor: Q-p. In *International Conference on Parallel Processing*, August 1987.

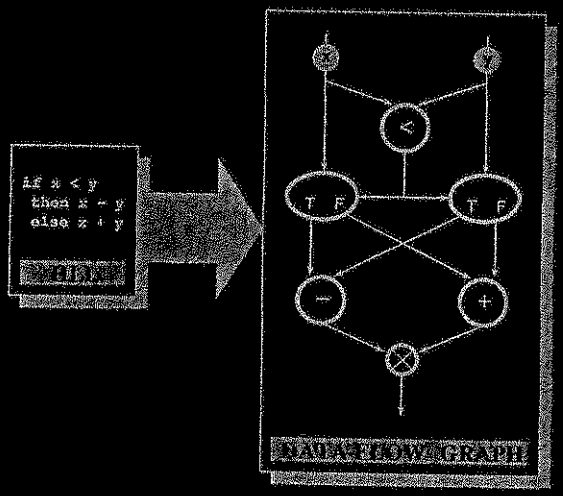
[13] G. Papadopoulos. *Implementation of a General Purpose Dataflow Multiprocessor.* PhD thesis, Department of Electrical Engineering and Computer Science, MIT, December 1987.

[14] M.W. Rawling. *Implementation and Analysis of a Hybrid Dataflow System.* Master's thesis, Department of Communication and Electronic Eng., RMIT, March 1988.

[15] T. Shimada, K. Hiraki, K. Nishida, and S. Sekigucki. Evaluation of a prototype data-flow processor of the SIGMA-1 for scientific computations. In *13th Annual International Synposium on Computer Architecture*, pages 226–234, 1986.

[16] A.J. Young. Implementation of a multistage network for interconnecting a dataflow multiprocessor. Technical Report TR 112 077 R, RMIT, November 1988.

JEAN-LUC GAUDIOT ■ LUBOMIR BIC




# ADVANCED TOPICS IN DATA-FLOW COMPUTING