The Proto-Architecture of WOLF, a Dataflow Supercomputer

A. Garcia Neto* . C. A. Ruggiero †

Grupo de Instrumentação e Informática
Departamento de Física e Ciência dos Materiais
Universidade de São Paulo, Brasil

Abstract

This article presents the proto-architectecture studied to define the characteristics of the Wolf processor, a proposal for the implementation of a high-performance supercomputer based on the dataflow paradigm. In order to place this proposal in the context of research in dataflow architecture, a survey of the main efforts in this area carried out in Europe, North America and Japan is done.

Keywords: Parallel Architectures, Dataflow, MIMD Implementations.

1 Introduction

The Wolf architecture is loosely based of the Manchester Multi-ring Dataflow Machine(MMDM) [26, 27, 59], incorporating many of the lessons learned since the implementation of that prototype ten years ago and the independent studies and modelling done elsewhere [20, 21].

The Wolf architecture is based on the following premises:

- Dataflow machines address elegantly and efficiently some difficult problems ocurring in other parallel machines, such as process allocation, load distribution, linear speed-up, memory contention and access latency. Dataflow machines are also easy to program, can expose and exploit parallelism without the programmer's help and has a well-defined mechanism to avoid the exposition of excessive parallelism [42, 51].
- The von Neumann architecture can process algorithms of low parallelism more efficiently than dataflow architectures [18, 30].
- SIMD architectures are more efficient than dataflow architectures to exploit parallelism derived from regularity in data structures [31].
- The RISC approach allows the implementation of highly efficient sequential processors [41, 57].
- Dataflow architectures are profligate in memory usage [18, 19].
- The circuits needed to implement Matching Memories using content-addressable components, common to most dynamic dataflow architectures, are complex and wasteful of silicon.
- Parallelism should be extracted out of algorithms without burdening the programmer with architectural details of the parallel machine.

* *alvaro@uspfsc.ifqsc.usp.ansp.br*
† *toto@uspfsc.ifqsc.usp.ansp.br*

2 Dataflow Architectures

Most dataflow implementations are based either on the the *ring* and the *network* organization, typified by the now classical designs of Manchester and of the MIT. This section places the development of the Wolf architecture in the context of previous work in dataflow.

2.1 Research in Europe

2.1.1 Manchester

The Manchester Dataflow Research Group has proposed the ring-based architecture [24, 25, 26, 27, 59] of figure 1.

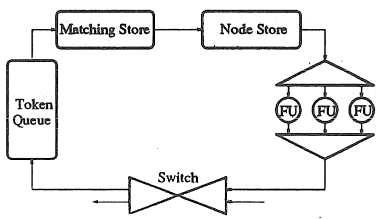

Figure 1: The Manchester Architecture

The MMDM is a collection of rings connected by a routing network. Within each ring there are units to smoothe the traffic (token queue), to implement the firing rule (the matching unit), to store the graph (the node store) and to execute the dataflow instructions (the functional units). The MMDM implements a dynamic tagged model. The programming language of the prototype is SISAL.

2.1.2 DTN

The DTN Dataflow Computer is a commercially available graphics-oriented workstation with 32 PE static dataflow engines by the Dutch company Dataflow Technology Netherland [58]. The workstation uses the ImPP [54] developed by NEC, coupled to a VME Unix system and a graphics subsystem with four 64 MOPS systolic arrays.

2.2 Research in North America

2.2.1 The VIM Machine

The Dennis group at MIT pioneered many of the basic dataflow concepts [16, 15, 17]. Research has continued from 1968 to date, currently aimed at the construction of a 1 GFlop static dataflow machine for large numerical computation. The VIM machine, shown in figure 2, is a collection of routing networks, cell blocks (CB), functional units (FU) and array memories (AM).

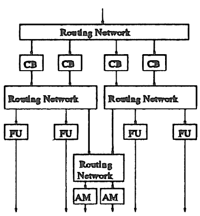

Figure 2: The VIM architecture

The cell blocks store the data dependency graphs and the data structures needed to implement the dataflow firing rule and termination acknowledgements, and execute simple operations such as duplicating tokens. The functional units consume the data scheduled by the cell block. Structured data are stored in the array memories. The routing network tolerates latency. The programming model for VIM is the VIMVAL [28], and extension of the VAL language [14, 10] that treats functions as first class objects, that can be passed and returned as parameters.

2.2.2 The MIT-Tagged Machine

This research was begun at Irvine in 1975 and is continued at the MIT by Arvind's Grup, evolving into the proposal of the MTTDA (the MIT Tagged-Token Dataflow) [3, 5, 6], with the architecture of figure 3:

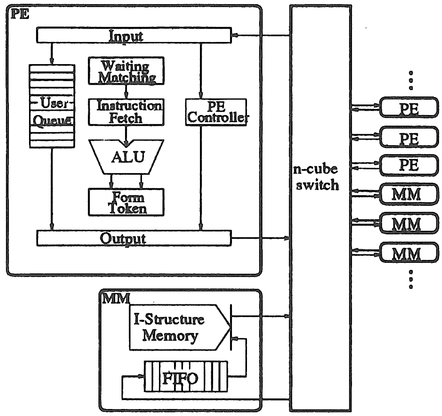

Figure 3: The MTTDA architecture

The MTTDA is an dynamic dataflow asynchronous machine with 64 PEs connect using a n-cube communication network and *I-structure nodes* that uses the tagging concept to enable more than one instantiation of a routine to share the same data dependency graph. The I-structure node stores static data and supports out-of-order reads and writes. A 256 board machine rated at 1 GIPS is under construction. The programming language for the machine is the dataflow language Id. The machine is aimed at both symbolic and numerical computations.

2.2.3 The Monsoon Machine

The Monsoon architecture [40] is a multithreaded one-address machine incorporating the Explicit Token Store (ETS) concept [13]. A Monsoon machine comprises a number of pipelined processors and I-structure memories connected by a routing network, as shown in figure 4.

The ETS concept requires that the frame pointer be local to a processor, and thus, activation names are restricted to the PE to where they were allocated, and execute entirely there. Because of the ETS, the firing rule is implemented as a indexed direct fetch to the frame store, which controls the arrival of partners.

The pool of tokens waiting to be processed is distributed, implemented as a two queues within the PE. The higher priority queue the *System Queue* FIFO, the lower priority queue is the *User Queue* stack. A PE ha a pipeline with the following stages: Instruction Fetcher, Effective Address Calculator, Presence Bits Operator (communication with the Presence Bits Memory), Frame Operator (communicating with the Frame Memory), ALU (three stages, including New Tag Calculator and communicating

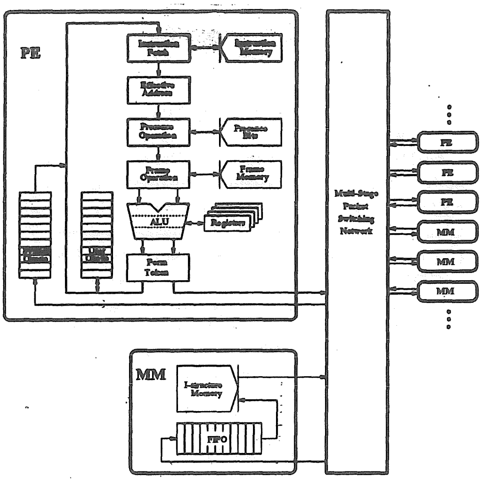

Figure 4: The Monsoon architecture

with a Register Bank), and Token Formator. A path has been provided to recirculate one of the result tokens back to the pipeline.

A 4 MIPS single PE prototype has been operational since 1988, executing a kernel SO written in Id. A joint MIT/Caltech/Motorola collaboration is developing a 10 MIPS, 10 MFLOPS single board PE with VME interface using 8 CMOS arrays with 10,000 gates [4]. The inter-PE routing network is implemented with PaRC 4x4 switching ICs [32]. A single PE Unix workstation running the Id dataflow language [38] is expected to be launched by Motorola in 1992. A system with 8 PEs, 8 Memory Modules, 8 Switch Modules and 4 Unix I/O Modules is also expected to be operational by then.

2.2.4 Other architectures

DDM1: The Data Driven Machine was designed at Borroughs in 1975. It is a colourless dynamic dataflow design, using FIFO queues to distinguish between different instantiations of graph reutilization. It has a topology of an octary tree connecting PEs containing an Agenda Queue, an Atomic Memory, an Atomic Processor and a Switching module.

TDDP: A Distributed Data Processor designed at Texas Instruments in 1976 to investigate the feasibilitiy of a static high performance dataflow design implementing a model that does not require confirmation of execution [11, 12]. A prototype using TTL logic was operational by 1978 using four PEs connected as a ring, using ADA as a programming language.

Epsilon-1, developed at Sandia National Laboratories as an evolution ot the DFAM project, was a test-bed dataflow processor that attained sustained uniprocessor performance comparable to contemporary mini-supercomputers [23, 22]. It uses *direct match*, a technique that enables single-cycle access to the matching unit implementing the dataflow firing rule, and uses *repeat-on-input*, a technique that minimizes the overhead associate with limited fan-out of dataflow instructions in most implementations.

Epsilon-2 is a successor of Epsilon-1 that fully supports the dynamic dataflow model. It is a variable-grain machine using a hybrid scheduling mechanism that enables both sequential and dataflow execution. The Epsilon-2 architecture comprises a number of PEs connected by a routing network, as shown in figure 5.

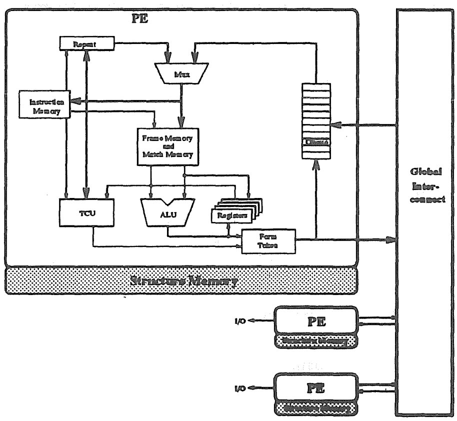

Figure 5: The Epsilon-2 architecture

The PE architecture draws from the Epsilon-1 and Monsoon PE architectures, and executes code compiled in Id, Sisal and Fortran. The PEs have support for I-structures, and load balancing is implemented by biasing the routing network as to direct allocation to the least loaded PE, in a scheme reminiscent of EM-4.

2.3 Research in Japan

2.3.1 SIGMA-1

Developed at the Eletrotechnical Laboratory (ETL/MITI), the SIGMA-1 project studies the feasibility of constructing a dynamic dataflow computer for numerical computation of realistic size [29, 49]. This project started in 1982 and was completed by 1988. The computer has over 128 synchronous PEs and sructure memories grouped in clusters and connected by a dual-level hierarchical network, as shown in figure 6.

The hardware is a synchronous microprogrammed CISC processor based on a 10,000-gate semi-custom

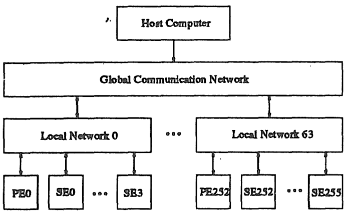

Figure 6: The Sigma-1 architecture

LSI, with 27 chips per board. Theoretical peak performance is 423 MFlops, 640 MIPS for a 128 node processor. Performances over 170 MFlops have been reported [45, 51]. Instructions have 40 bits for tagged data, and can generate results to up to three other instructions. SIGMA-1 supports DFC-2, a C-like language with single-assignment restrictions.

2.3.2 EM-4

The fourth generation of symbolic processing machines of ETL, EM-4 is a variable grain dynamic dataflow machine using the concept of strongly connected arc [34, 51]. A single chip RISC dataflow processor, called EMC-R, was built in 1988 with 50,000 gate semi-custom LSI technology [44]. A EMC-R encompasses the basic dataflow function: packet switching, input buffering, instruction fetch, operand matching, and instruction execution. An 80 node prototype is operational since 1990, as shown in figure 7.

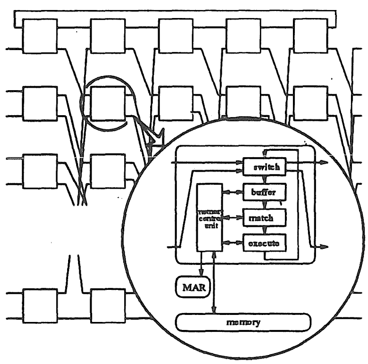

Figure 7: The EM-4 architecture

The theoretical peak performance of the prototype is 1.1 GIPS for a 80 node processor, 14.6 GIPS for a 1024 node [51].

2.3.3 Other Projects in Japan

Many other dataflow machines have been proposed, built or simulated such as:

NEDIPS developed at Nippon Telegraph and Telephone Corporation (NTT) in the late 1970s, is a hardwired static dataflow architecture dedicated to image processing with an extension to support multiple tokens per arc.

ImPP, the Image Pipelined Processor developed at NTT in the early 1980s is an implementation of NEDIPS in a single chip integrated circuit, intended to be used as a building block to larger systems. The ImPP is commercially available since 1985 [58].

EDDY, developed at NTT, is an array of 16 nodes with *nearest neighbour* connectivity. Each node has two Z8000 microprocessors. It uses VALID as a programming language. The project started in 1980, the prototype ran in 1983 [53] and was reported finished in 1986.

TopStar of Tokio University is a Z80 microprocessor-based prototype with 16 PEs, exploiting dataflow at the inter-procedure level [48] aimed at the recognition of printed Chinese character patterns. The project started in 1978 and finished by 1982, and is followed by the TOPSTAR-II and, subsequently, the PIE projects. THe porgramming languages used are Lisp and Prolog.

DDDP of Oki Electric Industry Co is a four processor, fixed-logic small scale prototype for numerical applications, started in 1980 and finished in 1982 [33].

DFM at NTT. A feasibility study for a dataflow processor for symbolic manipulation and list-processing, with lazy cons strategy [1, 2, 39]. Project started in 1982, a two-processor prototype ran in 1985. The high level language is VALID. A CMOS LSI version, DFM-II was started in 1985.

PIM-D of Oki Electric Industry Co. is a 16 processor, small-scale prototype for numerical applications, started in 1982 and operational in 1984 Project. It uses PROLOG as a programming model.

EM-3 of ETL is a Lisp machine, with 16 PEs based on the Motorola 68000 microprocessor. The project started in 1982 and the prototype is operational since 1985 [60]. It uses EMLISP, a single-assignement LISP subset.

DFNDR-1 at Gunma University, is a four processor, microprocessor-based (6809), small scale prototype for performance evaluation [47]. It used static dataflow, and the project was reported finished by 1986.

TIP, the *Template-based Image Processor* is a commercial static dataflow integrated circuit developed by NEC for image processing [54], available since the early 1980s.

Q-v1 a joint development of Osaka University, Sharp, Mitsubishi, Sanyo and Matsushita, directed to signal processing and image processing [56]. It is an 8 PE prototype using a five chip set based on the Manchester design, implementing a dynamic tagged dataflow model [35] and using the elastic pipeline approach [36].

3 The Wolf Architecture

3.1 Known Problems

Many lessons have been learned during the implementation of the MMDM:

- The circular shape of the pipeline increases latency time and causes traffic fluctuation.
- The number of stages of the MMDM pipeline is larger than required, and hampers performance.
- The asynchronous implementation of the modules causes delays in the pipeline and makes project debbuging very complex.
- The fixed data word size of 32 bits is inadequately small for some applications and excessively large for others
- The granularity of the MMDM instructions seems to be finer than it should be.
- The fixed granularity of the MMDM hampers research about optimal granularity level in dataflow processors.
- There is redundancy in several token fields, unnecessarily increasing the matching space.

3.2 A Pathological Example

The data dependency graph of figure 8, a typical code for parameter-passing in function calls of SISAL programs compiled for the MMDM [37] shows some of the disadvantages of the dataflow approach [7, 8, 9].

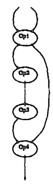

Figure 8: Pathological code.

In this code, a sequential ordering of instructions is enforced by the data dependency. The von Neumann architectures, due to their implicit sequentiality, are much more efficient in these cases. In a typical dataflow processor, no information about the scheduling of instruction `Op2` can be made at the time of the triggering of the `Op1` instruction. However, it is easy to see that the firing rule that schedules `Op1` has also scheduled, sequentially, `Op2`, `Op3` and `Op4`. Since a typical dataflow processor cannot see this implicit scheduling, the token with the result from `Op1` will have to go through an entire pipeline round for each of one of the instructions `Op2`, `Op3` and `Op4`.

This inneficiency is made worse by the fact that no instruction following `Op1` needs any further input data. The single output from the previous instruction is their sole input. In the MMDM terminology, these unary tokens are called BYPASSes, but are still sent to the matching unit, where a dummy firing mechanism (which always yields a match) is used. Simulations show that up to 60% of all executed instructions can have unary inputs [55].

Another problem detected in this situation is the creation of "short-lived" tokens, such as the intermediary result of `Op1` needed to trigger `Op4`. In a typical dataflow processor, this intermediary result is stored in the matching unit and almost immediately retrieved. This pointless exercise, coupled with the traffic of unary instructions, making the matching unit the slowest and most critical unit of the pipeline.

3.3 Working Description

The proto-Wolf Supercomputer consists of the interconnected units shown in figure 9. In the following working description, the nomenclature from the MMDM has been retained where possible.

A token represents a piece of data on an arc of the data-dependency graph being executed. Tokens are inserted into the machine via the input datapath of the Collecting Network. Answers are sent to the host computer through the output datapath of the Distributing Network.

From the Collecting Network, tokens pass through the Token Queue to the Matching Unit. The Matching Unit is responsible for matching operands so that both operands of a dyadic operator are made available to the subsequent units. It combines the output from the Data Memory to forms a Group Package, a pair of normal tokens directed to the same node in the data-dependency graph and with the same label. The Group Package is distributed through the Distributing Network to one an available parallel Node Store. If no Node Store is available, the Group Package is grabbed by the Overflow Unit for latter distribution. The Node Store holds the details of the node which the pair of operands have reached. It proceeds an Executable Package containing all the information required to execute the node, and forwards it to its associated Functional Unit. The results produced by the Functional Unit can be either *free* or *chained*. Free results are passed to the Collecting Network, and thence round the ring. Chained results remain in the Functional Unit internal registers, to be, at some stage, used as operands to subsequent

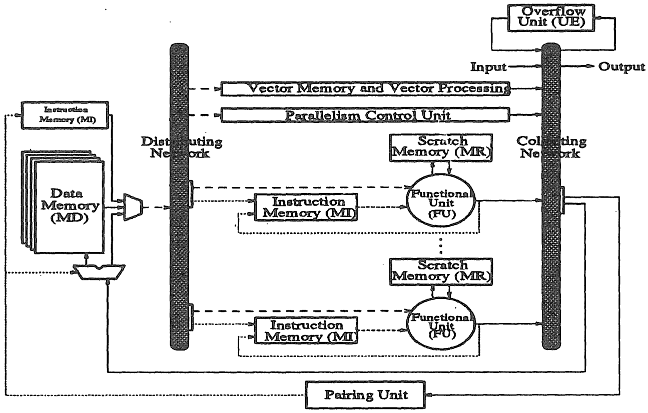

Figure 9: The Wolf Dataflow Architecture

operations. If the Collecting Network cannot forward the received tokens, the Overflow Unit stores them to form a pool of available data for latter distribution

One of the details held by the Node Store is wether a given node is part of a chain or not. When an Executable Package belonging to a chained node is produced, the Node Store begin an extra fetch cycle, using the destination of the recently produced package as address. This spontaneous fetch cycle progresses in parallel, while the Functional Unit consumes the aforementioned Executable Package. Thus, when the consumption is finished, a new Executable Package is produced from the Node Store ready to be executed, using the results of the previous package as input.

Connected to the Collection and Distributing Networks are one or more Structure Store units, used to store data. The Structure Store allocates space on request, and supports a reference count mechanism, automatically reclaiming the space if the reference count is ever reduced to zero. The write and reads of a location may be issued in any order, the store deferring within itself requests for access to a value which is not yet present. It is, however, an error to write to a location which is already occupied. Besides reading from a location, which is non-destructive, it is possible to extract from location, leaving it empty. Messages to the Structure Store are constructed by Structure Store orders executed in the Functional Units.

Tokens are inserted into the machine via the input datapath of the Collecting Network. Answers are sent to the host computer through the output datapath of the Distributing Network.

From the Collecting Network, tokens pass through the Token Queue to the Matching Unit. The Matching Unit is responsible for matching operands, where necessary, so that both operands of a dyadic operator are made available to the subsequent units. It combines the output from the Data Memory to form a Group Package, a pair of tokens directed to the same node in the data-dependency graph and with the same label. The Group Package is distributed through the Distributing Network to any available parallel Node Store. The Node Store holds the details of the node which the pair of operands have reached. It proceeds an Executable Package containing all the information required to execute the node, and forwards it to its associated Functional Unit. The results produced by the Functional Unit can be either *free* or *chained*. Free results are passed to the Collecting Network, and thence round the ring. Chained results remain in the Functional Unit internal registers, to be at some stage used as operands to subsequent operations.

The Node Store holds whether a given node is part of a chain or not. When an Executable Package belonging to a chained node is produced, the NS starts autonomously an extra fetch cycle, using the destination of the recently produced package as address. This spontaneous fetch cycle progresses in parallel, while the Functional Unit consumes the aforementioned Executable Package. Thus, when the consumption is finished, a new Executable Package is produced from the Node Store ready to be executed, using the results of the previous package as input.

3.4 Architecture Characteristics

3.4.1 Variable Granularity

The main characteristic of the Wolf processor is its ability to execute code as either a dataflow processor or as a von Neumann processor. This concept has already explored elsewhere [20, 30] in the context of variable granularity. The Wolf architecture, shown in figure 9, enables a stream of sequential instructions to be generated and presented to a Functional Unit. In this units, this stream of functions will successively operate upon the accumulator register, where the result of the previously executed instruction is still stored.

3.4.2 Locality

The stream of instructions is generated whenever a flag bit is planted by the compiler. The detection of these sequential chains in data dependency graphs is common practice in many situations, and does not pose special difficulties. Conversely, "short-lived" results can be easyly detected, and special LOAD and STORE instructions planted to store this result in a scratch memory. The resulting tokens can be directed directly to the Instruction Memory, where new streams of instructions are triggered. Thus, many disjoint groups of sequential instructions can be executed between the pair Functional Unit-Node Store before a reference to the global matching mechanism.

With this enforced locality, the issue of load distribution arises. In typical dataflow processors, the spread of fine grain instructions is sufficient to the ensure an efficient, transparent and almost cost-free balancing mechanism. With enforced locality, some processors might become overload while others idle. The Wolf architecture avoids this situation by providing separate paths for matching tokens and for unary tokens.

The Connecting Network that gathers the output from the Functional Units has four exit paths: One directs data outside the machine, two direct data to the matching mechanism and one sends data directly to the Distributing Network. This last path is provided to enable an overloaded Functional Unit to offload part of its traffic to some other Functional Unit. Each Functional Unit communicates to the Distributing Network its level of activity, and the network is capable of directing the traffic to the least busy Functional Unit. This approach was successfully employed in the SIGMA-1 dataflow computer [50, 52]

3.4.3 Parallelism Control and Vector Processing

The processing elements between the Distributing and Collecting Networks do not have to be homogeneous. One known characteristic of dataflow designs is its ability to expose more parallelism than required, flooding the machine with more (potentially) parallel actions than the number of available processing elements [43]. One solution for this problem is to limit the number of different instantiations of the data dependency graphs, limiting the creation of new *Activation Names* [42, 55] This is done by the Parallelism Control Unit, which also performs the recycling of these activation names by identifying instantiations that have already finished.

At this same level stands the Vector Processing Unit, which exploits parallelism in data structure regularity. In Wolf, these regular data structures do not circulate in the ring: Instead, pointers are processed, and, when the actual operation is due, an action is requested to the Vector Processing Unit. This aproach was suggested by Sargeant for the Structure Store, but never implemented [46].

3.4.4 Matching Unit and Data Memory

Since the operations remain constrained to a single Functional Unit, the enforced locality dispenses with large quantities of activation names. The implementation of the Parallelism Control Unit at the same hierarchical level of the Functional Units allows for effective Activation Name recycling, and the Vector Processing Unit dispenses with most indexes in tokens.

These factors concur for a large decrease in the size of the matching space, which becomes compatible to the sizes of current virtual memories. The Data Memory, can now be managed using the same proven techniques. In Wolf, it is a standard paged virtual memory, addressed by the Matching Unit.

The Collecting Network has two separate paths for the matching mechanism: one takes tokens to the Data Memory, the other takes addresses to the Matching Unit. The Matching Unit uses this address to verify if a pair to this token is waiting in the Data Memory. If there isn't a waiting pair, it generates a write signal which stores the incoming data in the required address. If a pair is already waiting, the Matching Unit generates a read signal and forwards the address. The Data Memory is responsible for joining the incoming data with its own stored data. Its output is a group of tokens with the same tag and destination address.

4 Implementation Details and Aims

- Variable word length, from 32 to 96 bits.
- Reduced token size (as compared to the MMDM): 26, 56, 152 and 164 bits.
- Variable granularity: Fine grain, stream of instructions and thick grain.
- Functional Units implemented using RISC processors with scratch memory.
- Matching Unit independent of Data Memory. Random addressable Data Memory.

- Matching of up to four tokens.
- Instruction Memories directly coupled to the Functional Units, capable of generating instruction streams using *prefetch*.
- Separate data paths for bypass and matching tokens.

The units of Wolf exchange information solely by tokens, which are subdivided in ***fields***. The widths of the inter-module busses is one of the critical points of most dataflow designs. The size of the token fields used in Wolf, which determine the width of the busses, are shown in table 1.

Field	Name	Function	Size
$\mathcal{A}$	Activation	Identification of activation	10
$\mathcal{C}$	CodOp	Operation Code	6
$\mathcal{D}$	Data	Data	32
$\mathcal{E}$	Matching	Number of matching tokens	2
$\mathcal{N}$	Node	Address of instruction in program	12
$\mathcal{T}$	Type	Data type	4

Table 1: Fields of the tokens

References

[1] M. Amamiya and R. Hasegawa. Dataflow Computing and Eager and Lazy Evaluation. *J. New Generation Computers*, 2(8):105–129, 1984.

[2] M. Amamiya, R. Hasegawa, O. Nakamura, and H. Mikami. Implementation and Evaluation of a List Processing-Oriented Dataflow Machine. In *Proc. 13th Annual Int. Symp. Computer Architecture*, 10–19. IEEE/ACM, 1986.

[3] Arvind and D. Culler. Dataflow Architectures. CSG Memo 294, LCS, MIT, Cambridge, Mass., USA, 1986.

[4] Arvind, M. L. Dertouzos, R. S. Nikhil, and G. M. Papadopoulos. PROJECT DATAFLOW, a Parallel Computing System Based on the Monsoon Architecture and the Id Programming Language. Technical Report CSG Memo 285, LCS, MIT, 1988.

[5] Arvind, V. Kathail, and K. Pingaley. A Dataflow Architecture With Tagged Tokens. CSG Memo 174, LCS, MIT, Cambridge, Mass., USA, Sep 1980.

[6] Arvind and R. S. Nikhil. A Dataflow Approach to General-Purpose Parallel Computing. CSG Memo 303, Lab. Computer Science, MIT, Cambridge, Mass., USA, Jul 1989.

[7] A. P. W. Böhm and J. Sargeant. Efficient Dataflow Code Generation for SISAL. Internal Report 2nd issue, DCS, Univ. Manchester, England, Sep 1985.

[8] A. P. W. Böhm and J. Sargeant. Efficient Dataflow Code Generation for SISAL. In M. Feilmeier, G. Joubert, and U.Schendel, ed., *Parallel Computing 85*, 339–344. North Holland Publishing Co, Jun 1986.

[9] A. P. W. Böhm and J. Sargeant. Code Optimisations for Tagged-Token Dataflow Machines. *IEEE Trans. Computers*, 38(1):4–14, Jan 1989.

[10] J. D. Brock. Operational Semantics of a Data Flow Language. Technical Report TM–120, LCS, MIT, Cambridge, Mass., USA.

[11] M. Corsish. The TI Dataflow Architectures: The Power of Concurrency for Avionics. In *Proc. 3rd Conf. on Digital Avionics Systems*, 19–25, Forth Worth, TX, USA, Nov 1979. IEEE.

[12] M. Corsish, D. W. Wogan, and J. C. Jensen. The Texas Instruments Distributed Processor. In *Proc. Louisiana Computer Exposition*, 189–193, Lafayette, LA, USA, Mar 1979.

[13] D. E. Culler and G. M. Papadopoulos. The Explicit Token Store. *J. Paralle and Distributed Computing*, 10(4), Jan 1991.

[14] J. B. Dennis. First Version of a Dataflow Procedure Language. In *Proc. Colloque sur la Programacion. v.19 (Lecture Notes in Computer Science)*, 362–376. Springer-Verlag, Berlin, 1984.

[15] J. B. Dennis. Modelling the Weather with a Dataflow Supercomputer. *IEEE Computer*, 3(7):592–603, 1984.

[16] J. B. Dennis. Models of Data Flow Computation. In *Proc. 1984 IEEE CompCon*. IEEE Computer Society, 1984.

[17] J. B. Dennis. Data Flow Computation. In M. Broy, editor, *NATO ASI Series, v.F14, Control Flow and Data Flow: Concepts of Distributed Programming*, 346–397. Springer-Verlag, Berlin, 1985.

[18] J. F. Foley. Manchester Dataflow Machine: Benchmark Test Evaluation Report. Internal Report UMCS–89–11–1, DCS, Univ. Manchester, England, Nov 1989.

[19] D. D. Gajski, D. A. Pádua, D. J. Kuck, and R. H. Kuhn. A Second Opinion on Data Flow Machines and Languages. *IEEE Computer*, 15(2):58 – 69, Feb 1982.

[20] D. Ghosal and L. W. Bhuyan. Analytical and Architectural Modifications of a Data Flow Computer. *Computer Architecture News*, 15(2):81–89, Feb 1987.

[21] D. Ghosal and L. W. Bhuyan. Performance Evaluation of a Dataflow Computer. *Trans. Computers*, 39(5):615–627, May 1990.

[22] V. G. Grafe and J. E. Hoch. Implementation of the Epsilon Dataflow Processor. In *Proc. 23rd Hawaii Int. Conf. on System Sciences*, 1990.

[23] V. G. Grafe and J. E. Hoch. The Epsilon-2 Multiprocessor System. *J. Parallel and Distributed Computing*, 10(4):309–318, Dec 1990.

[24] J. R. Gurd et al. Fine-Grain Parallel Computing: The Dataflow Approach. Dataflow Group internal document in preparation, DCS, Univ Manchester, England.

[25] J. R. Gurd, C. C. Kirkham, and W. Böhm. *The Manchester Dataflow Computing System*, v. 1 of *Special Topics in Supercomputing*. North-Holland, Jan 1987.

[26] J. R. Gurd, I. Watson, and J. R. W. Glauert. A Multilayered Dataflow Computer Architecture. Internal report, DCS, Univ. Manchester, Mar 1980.

[27] J. R. Gurd, I. Watson, and C. C. Kirkham. The Manchester Prototype Dataflow Computer. *Comm. ACM*, 28(1), Jan 1985.

[28] J. Herath, Y. Yamaguchi, N. Saito, and T. Yuba. Dataflow computing models, languages, and machines for intelligent computations. *IEEE Trans. Software Engineering*, 14(12):1805–1928, Dec 1988.

[29] K. Hiraki, T. Shimada, and K. Nishida. A Hardware Design of the SIGMA-1 — A Data Flow Computer for Scientific Computations. In *Proc. Int. Conf. on Parallel Processing*, 524–531. IEEE, 1984.

[30] R. A. Iannucci. A Dataflow/von Neumann Hybrid Architecture. Technical Report TR–418, LCS, MIT, Cambridge, Mass., USA, 1988.

[31] R. N. Ibbet. *The Architecure of High Performance Computers*. The MacMillan Press, London, first edition, 1982.

[32] C. F. Joerg. Design and Implementation of a Packet Switched Routing Chip. Master's thesis, Dept. Electrical Eng. and Computer Science, MIT, Cambridge, Mass., USA, 1988.

[33] M. Kishi, H. Yasuhara, and Y. Kawamura. DDDP: A Distributed Data Driven Processor. In *Proc. 10th Annual Int. Symp. Computer Architecture*, 236–242. IEEE, 1983.

[34] Y. Kodama, S. Sakai, and Y. Yamagughi. A Prototype of a Highly Parallel Dataflow Machine EM-4 and Its Preliminary Evaluation. ETL Research Memorandum, Eletrotechnical Laboratory, 1990.

[35] S. Komori, K. Shima, S. Miyata, T. Okamoto, et al. The data-driven microprocessor. *IEEE Micro*, 9(3):45–60, Jun 1989.

[36] S. Komori, H. Takata, T. Tamura, F. Asai, T. Ohno, O. Tomisawa, T. Yamasaki, K. Shima, et al. An Elastic Pipeline Mechanism by Self-Timed Circuits. *J. Solid-State Circuits*, 23(1):111–117, Feb 1988.

[37] J. R. McGraw. SISAL: Streams and Iteration in a Single Assignement Language, Language Reference Manual, Version 1.2. Technical Report M–146, Lawrence Livermore National Laboratory, Mar 1985.

[38] R. S. Nikhil. Id (version 88.0) Reference Manual. Technical Report CSG Memo 284, LCS, MIT, Mar 1988.

[39] S. Ono, N. Takahashi, and M. Amamiya. Optimized Demand-Driven Evaluation of Functional Programs on a Dataflow Machine. In *Proc. Int. Conf. on Parallel Processing*, 421–428, 1986.

[40] G. M. Papadopoulos. Implementation of a General Purpose Dataflow Multiprocessor. PhD Thesis, Technical Report TR 432, LCS, MIT, Cambridge, Mass., USA, Sep 1988.

[41] D. A. Patterson and C. A. Sequin. A VLSI RISC. *IEEE Computer*, 15(9):8–21, Sep 1982.

[42] C. A. Ruggiero. Throttle Mechanisms for the Manchester Dataflow Machine. Technical Report Series UMCS-87-8-1, DCS, Univ. Manchester, England, Jul 1987.

[43] C. A. Ruggiero and J. Sargeant. Control of Parallelism in the Manchester Dataflow Machine. In *Proc. ACM Int. Symp. Functional Programming Languages and Computer Architecture*, 1 – 17, Oct 1987.

[44] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba. An Architecture of a Dataflow Single Chip Processor. In *Proc. 16th Annual Int. Symp. Computer Architecture*, 46–53, Jerusalem, Israel, Jun 1989. IEEE/ACM.

[45] M. Sakurai. Data-flow demon; MITI masters 640 MIPS. *Electronic Engineering Times*, (481):45–46, Apr 1988.

[46] J. Sargeant. Efficient Stored Data Structures for Dataflow Computing. Technical Report Series UMCS-85-8-2, DCS, Univ. Manchester, England, Aug 1985.

[47] M. Sowa and T. Murata. A Data FLow Computer Architecture with Program and Data Memories. *IEEE Transacitions on Computers*, C-31(9):820–824, 1982.

[48] T. Suzuki, K. Kurihara, H. Tanaka, and T. Moto-oka. Procedure. Level Data Flow Processing on Dynamic Structure Multimicroprocessors. *J. Information Processing*, 1(5):11–16, 1982.

[49] T. Shimada and K.Hiraki and K. Nishida. An Architecture of a Data Flow Machine and Its Evaluation. In *Proc. COMPCON 84 (Spring)*, 486–490. IEEE, 1984.

[50] T. Shimada and others. Evaluation of a Prototype Data Flow Processor of the SIGMA-1 for Scientific Computations. In *Proc. 13th Int. Symp. Computer Architecture*, 226–234. IEEE/ACM, Jun 1986.

[51] T. Shimada T. Yuba et al. Dataflow Computer Development in Japan. *ACM*, 140–147, 1990.

[52] T. Shimada T. Yuba, K. Hiraki et al. The SIGMA-1 Dataflow Computer. Internal report, Eletrotechnical Laboratory, MITI, Japan.

[53] N. Takahashi and M. Amamiya. A Data Flow Processor Array System — Design and Analysis . In *Proc. IEEE 10th. Annual Int. Symp. Computer Architecture*, 243–250. IEEE, 1983.

[54] T. Temma, S. Hasegawa, and S. Hanaki. Dataflow Processor for Image Processing. *Proc. on Mini and Microcomputers*, 5(3):52–56, 1980.

[55] Y. M. Teo. Concurrency Control in the Multi-Ring Manchester Dataflow Machine. Master's thesis, Univ. Manchester, England, Sep 1989.

[56] H Terada, H Nisikawa nd K. Asada, S. Matsumoto, S. Miyata, S. Komori, and K. Shima. VLSI Design of a One-Chip Data-Driven Processor: Q-v1. In *Proc. Fall Joint Computer Conf.*. ACM/IEEE, 1987.

[57] F. J. Valente. Estudo de arquiteturas risc. Master's thesis, DFCM, IFQSC, Universidade de São Paulo, 1991.

[58] A. H. Veen and R. van den Born. The RC Compiler for the DTN Dataflow Computer. *J. Parallel and Distributed Computing*, 10(4):319–332, Dec 1990.

[59] I. Watson and J. R. Gurd. A Practical Dataflow Computer. *IEEE Computer*, 15(2), Feb 1982.

[60] Y. Yamagushi, K. Toda, J. Herath, and T. Yuba. EM-3: A LISP-Based Data-Driven Machine. In *Proc. Int. Conf. on Fifth Generation Computer Systems*, 524–532. ICOT, 1984.