AN ABSTRACT OF THE THESIS OF

Roger L. Traylor for the degree of Master of Science in Electrical and Computer Engineering presented on May 3, 1991.
Title: A Token Caching Waiting-Matching Unit for Tagged-Token Dataflow Computers

Abstract approved: ____Redacted for Privacy______
Bella Bose

Computers using the tagged-token dataflow model are among the best candidates for delivering extremely high levels of performance required in the future. Instruction scheduling in these computers is determined by associatively matching data-bearing tokens in a *Waiting-Matching Unit* (W-M unit). At the W-M unit, incoming tokens with matching contexts are forwarded to an instruction while non-matching tokens are stored to await their matching partner.

Requirements of the W-M unit are exacting. Necessary token storage capacity at each processing element (PE) is presently estimated to be 100,000 tokens. Since the most often executed arithmetic instructions require two operands, the bandwidth of the W-M unit must be approximately twice that of the ALU. The contradictory requirements of high storage capacity and high memory bandwidth have compromised the M-W units of previous dataflow computers limiting their speed.

However, tokens arriving at a PE exhibit strong temporal locality. This naturally suggests the use of some caching technique. Using a recently developed CAM memory structure as a base, a token caching scheme is described which allows rapid, fully associative token matching while allowing a large token storage capacity.

The key to the caching scheme is a fast and compact, articulated, first-in, first-out, content addressable memory (AFCAM) which allows associative matching and garbage collection while maintaining temporal ordering. A new memory cell is developed as the basis for the AFCAM in an advanced CMOS (Complementary Metal Oxide Semiconductor) technology. The design of the cell is discussed as well as electrical simulation results, verifying its operation and performance.

Finally, estimated system performance of a dataflow computer using the caching scheme is presented.

A Token Caching Waiting-Matching Unit for Tagged-Token Dataflow Computers

by

Roger L. Traylor

A THESIS
submitted to
Oregon State University

in partial fulfillment of
the requirements for
the degree of
Master of Science

Completed May 3, 1991

Commencement June 1991

APPROVED:

Redacted for Privacy

Professor of Computer Science in charge of major

Redacted for Privacy

Head of department of Electrical and Computer Engineering

Redacted for Privacy

Dean of Graduate School

Date thesis is presented May 3, 1991

Presented by Roger L. Traylor

Acknowledgements

I owe my deepest gratitude to the Lord Jesus Christ for helping me with this thesis. He is the author of all that is good in this paper and in me. To Him be all honor and glory. Amen.

I would like to thank Jane, my loving wife for supporting me during the preparation of this thesis. I am in eternal debt to you. I would also like to thank my sons Andrew and William for giving up time with Dad to let me work on this project.

I would like to thank my Mom and Dad who encouraged my education while not pushing too much. I especially appreciate my Dad who encouraged me build and experiment with everything from organic gardening to motorcycles to electronics. I miss you Dad.

I would also like to extend my appreciation to Jesus Pena who first helped me get the initial designs of the AFCAM cell running. Thanks also to Dave Dunning for all the weeknights he spent away from Lisa to help me get the AFCAM cell right. Without you two guys, the AFCAM would have never flown.

Thanks to Dr. Bella Bose whose kindness and support made this whole process much easier. Thanks also to Dr. Joe Minne who first introduced me to dataflow architectures and stimulated my thinking about their limitations.

And finally, thanks to Intel Corporation for making available the tools I needed to do my simulations. Thanks also to my colleagues at Intel who provide an endless source of ideas and enthusiasm.

Table of Contents

List of Illustrations

A Token Caching Waiting-Matching Unit for Tagged-Token Dataflow Computers

I. Introduction

Computers using the tagged-token dataflow model are among the best candidates for delivering extremely high levels of performance required for future computing needs [Lerner 84]. However, to this date a high performance hardware realization of the abstract dataflow model has not been achieved. One major reason for this is the way in which the dataflow instruction scheduling is implemented in hardware.

Instruction scheduling in tagged-token dataflow computers is determined by associatively matching data-bearing tokens in a *Waiting-Matching Unit* (W-M unit). At the W-M unit, tokens with matching contexts have their operands forwarded to an instruction while non-matching tokens are stored to await their matching partner. The complexity of the waiting-matching unit makes its design one of the major stumbling blocks in realizing a tagged-token dataflow computer [de Silva 83].

To successfully implement a practical, high performance dataflow M-W unit, two areas of understanding must be established. First, a thorough understanding of the dataflow computational model, especially the token matching semantics is required. Secondly, an understanding of real-world hardware structures and their particular limitations in implementing the W-M unit is necessary.

The bold opening statement of this thesis may be difficult for some to accept. However the dataflow model is the best understood, most thoroughly researched paradigm for creating parallel computer architectures [Lerner 84]. Furthermore, by examining current trends in computer technology, the foundation upon which this claim is based will become apparent. We will now examine this foundation.

1. Why Dataflow?

1.1 Limitations of Physics

To make integrated circuits and thus computers run faster, the physical size of devices within an integrated circuit are reduced. This is because smaller transistors are faster and increased integration avoids circuit delays encountered whenever signals leave a chip.

Unfortunately, we are beginning to reach physical limits as to how small devices within integrated circuits can be. Currently, MOS transistor channel lengths are about 1um. This can possibly be reduced by a factor of 10, but fundamental limits will most likely prevent a shrinking in size by a factor of one hundred [Seitz 84]. As feature sizes fall below 0.25 microns, problems such as tunneling through very thin oxide layers begin to appear. Also, statistical variations in the threshold voltage (V_{th}) can occur due to the small numbers of impurity ions present.

Even if a reduction in transistor size by a factor of one hundred could be realized, the additional on chip interconnects needed would drive down the active device density. Furthermore, narrowing the interconnections will not help. If an interconnect is shrunk, its cross sectional area and resistance increase quadratically. The capacitance per unit length however, remains approximately constant, thus increasing the propagation delay of the interconnect. Thus, as feature sizes fall, linear increases in operating speed become more difficult to achieve. Before long, further integration will yield diminishing returns in terms of performance for a given cost.

1.2 Current Parallel Architectures Limit Performance

It is widely accepted that the next step for increased computer performance is to exploit concurrency. The basic idea is to have many slower processing elements (PEs) work cooperatively on a problem instead of a single very fast CPU. However, the way in which the PEs cooperate to solve a problem today is far from optimal.

Parallel computers being built today basically fall into two different architectural categories; shared memory and distributed memory. Shared memory machines allow communication between processors by sharing a common memory. They synchronize by using semaphores or spin and test structures. Distributed memory computers typically communicate between processors by sending data (usually a message) through some type of communication network. The reception of a message acts as the synchronization mechanism. Both of these approaches have considerable drawbacks.

Shared memory computers have a bottleneck at the common memory used by processing elements. As the number of PEs increase, memory contention increases along with effective memory access time. Thus, beyond a few dozen PEs, asymptotic performance gains result. Using a local cache memory at each PE helps alleviate memory contention somewhat but the resultant cache coherency problems are tricky to resolve.

The amount of time it takes to send and receive a message is a weak point of message passing computers. Typical values of message latency for these computers is on the order of hundreds of microseconds [Pierce 90]. In order to keep each processing element busy doing useful work, the message latency must be masked by executing sufficiently large segments of code. This practice limits exploitable parallelism, and thus places an upper bound on maximum attainable speedup. If more levels of parallelism could be exploited, greater efficiency would result. As stated by Gurd and Treleaven [Treleaven 84]:

> "Intuitively, any computer architecture or language that supports a mechanism for operator level parallelism should support the higher levels. Thus it is necessary to start the exploitation of parallelism at the lowest levels."

In other words, if a computer architecture is able to expose and fully exploit parallelism in a program at the lowest levels, parallelism existing at higher levels will automatically be exploited. For example, if an architecture is able to expose instruction parallelism, parallelism at the function call level will naturally be exposed.

1.3 Poorly Constructed Parallel Programming Languages

Programming languages presently used on parallel computers are ill suited to the job of writing parallel programs. These languages are often modified sequential languages that have extra provisions "tacked on" for explicitly representing parallelism. (i.e. fork and join operations) The programmer is responsible for partitioning the problem into independent threads of computation. Thus, the programmer must be aware of the computer's architecture. Also, wherever program fragments execute in parallel and share data, the programmer is responsible for synchronization. Partitioning a program and synchronizing the parts is an error prone, difficult task. More importantly, this task nothing to do with solving the original problem. We clearly need a natural means for expressing parallelism and a machine that can efficiently execute the resulting algorithms. We need a new start.

2. The Dataflow Solution

Jack Dennis of MIT has identified three prerequisites for an ideal computer architecture of the future. First of all, the desired performance must be attainable at an acceptable cost. Secondly, the architecture of the computer must be well matched to available technology. Lastly, the computer must be reasonably easy to program in a way that allows the parallelism in a problem to be exploited by the hardware [Dennis 80]. The dataflow model of computing offers an approach which satisfies these prerequisites.

2.1 High Performance at Acceptable Cost

For applications requiring extremely high performance, dataflow computing may be the only reasonable approach. As the number of processors is increased, current parallel architectures exhibit asymptotic performance gains for fixed-size problems. A proportional or linear speedup would be ideal and is currently considered to be the maximum attainable. Dataflow architectures exhibit this linear speedup. In addition, for problems sufficiently parallel, dataflow computers can theoretically exploit all parallelism present in a program, providing the maximum possible performance [Arvind 82].

2.2 Computer Architecture Matched with Available Technology

Mead and Conway have formulated a number of guidelines for developing efficient VLSI architectures [Mead 80]. These guidelines stress that systems built with VLSI technology consist of many simple and identical computing elements, each with capabilities for processing, communication and memory. Within the system, there should be considerable locality of reference to reduce system-wide communications. Also, the system should consist of groups of elements that are functionally equivalent to a single element. The structure of a dataflow computer matches these guidelines well.

2.3 Easy to Program

Languages which dataflow computers naturally execute show much promise for improving the programmability of parallel machines [Treleaven 82]. These dataflow languages enable a program to be written in a way that preserves the structure of the problem's solution and hides the details of the machine from the programmer [Arvind 78]. The programmer need not know anything about the architecture to properly program the computer. The programmer can concentrate on problem solving without being bothered with unimportant architectural features of the computer [Ackerman 82].

3. Dataflow Research in Progress

In respect to "von Neumann history", dataflow architectures are in their infancy. The hardware complexity inherent to dataflow computers has greatly hindered their development. Now that VLSI technology is beginning to mature, dataflow machines may begin to reach their potential. Results thus far are very modest, yet encouraging, especially considering some machines have exhibited linear speedup for certain programs.

Briefly mentioned below are three dataflow computing projects which are presently underway. The selected projects represent the most developed tagged-token dataflow machines. The following short description of each project is to give the reader a feeling for the present state of dataflow computer development.

3.1 Electrotechnical Laboratory (Japan)

Several dataflow computers are being developed at the Electrotechnical Laboratory in Japan. Their work is being supported by the High-Speed Computing System for Scientific and Technological Use Project of the MITI. Sigma-1 consists of 128 PEs connected by a two level network. Each PE is designed to attain 3.3 MFLOPS and 5 MIPS peak. The total performance is 427 MFLOPS and 640 MIPS peak. 170 MFLOPS has been attained on a numerical integration problem. PE performance is actually 0.4 to 2.1 MFLOPS using benchmark programs extracted from the "Livermore loops" set [Hiraki 84]. This program is now nearing its completion.

The successor to Sigma-1 is a hybrid dataflow processor called EM-4. EM-4 will initially be a 80 processor machine running at 997 MIPS. The next machine will be a 1000 node version with floating point hardware with an estimated performance of 20 gigaflops [Kahaner 90].

The EM-4 designers felt that the pure dataflow model was too inefficient and thus built the processor with a hybrid dataflow model called the strongly connected block model. The main feature of the model is to take the instruction-level dataflow graph and collect the nodes together into single "strongly connected blocks". Each strongly connected block will then be executed on a RISC-style single processor with program counter and registers. Anything outside a block will be executed in a dataflow model where matching operands fire an instruction. Strongly connected blocks are macro nodes in a dataflow program graph and execute when all of their operands arrive [Lubeck 90].

The single chip processor has been designed and 5 chips fabricated. The processor along with 1 megaword of memory reside on a single board. The chips are 50,000 gate ASICs (Application Specific Integrated Circuits) manufactured by the Japanese division of LSI Logic in Tskuba. Sample programs are running on a 5 PE machine [Lubeck 90].

3.2 University of Manchester [England]

Twelve slow, microcoded ALUs, token queue, matching unit, and an instruction store have been operational since 1981. A number of reasonably large programs have been run on the machine with an average performance of 1.2 MIPS. Programs with large data structures have revealed that there is a need for specialized hardware to implement data structure storage. By adding processors, a nearly linear speedup was observed until limits imposed by the communications network and matching store were reached [Gurd 85].

3.3 MIT TTDA (Tagged-Token Dataflow Architecture)

This architecture developed around the theoretical work on the unraveling interpreter (U-interpreter) model developed by Dr. Arvind at the University of California at Irvine. His research is now continuing at MIT. His approach is to focus on the development of an entire dataflow system using extensive modeling techniques prior to building any hardware. Two soft prototypes have been implemented to serve as study vehicles. A simulator provides a detailed model of the machine, while a dataflow emulator is being developed to run on their emulation facility consisting of 38 TI Explorers and 8 Symbolics 3600's [Arvind 86].

3.4 Mitsubishi Electric

In 1991, researchers at Mitsubishi Electric's LSI R&D Laboratory described a 32-bit dataflow microprocessor that combines dataflow techniques with conventional superpipelining. Performance is estimated to be as high as 50 megaflops or 150 to 200 MIPS. The design is much less complex than current RISC or CISC designs with no more than about 700,000 transistors using 0.8 micron processing.

The processor is configured in a ring-like pipeline configuration. It has a ring interface, matching memory, data memory, program memory, and floating point unit into which data is fed in packet form, interleaved with other circulating data through the use of an asynchronous arbiter [Cole 91].

Because pipeline stalls created by memory latency does not occur in dataflow architectures, superpipelining works well. Each on-chip block is further sub-divided into six to twelve pipeline stages to allow faster operation. Rather than use high frequency clocks, self-timed design techniques are used to maintain the highest possible data flow rate [Bursky 91].

4. Scope and Method of Thesis

The scope of this thesis is to examine the problems with implementing a fast and efficient waiting-matching unit for a dynamic dataflow computer and to engineer a possible solution. The result is the design of a hierarchical waiting-matching unit with token cache that uses a novel memory cell.

Software and system service issues such as systems software, I/O, resource management, or memory protection are considered only where they directly effect the W-M unit hardware implementation or the understanding of the concepts involved.

A brief introduction to dataflow computing will be followed by a description of the fundamental concepts behind the dataflow computational model. The requirements for a W-M unit are then discussed. Finally the design for a token caching hierarchical W-M unit is described along with anticipated system performance.

II. Overview of Dataflow Computation

The purpose of this section is to explain dataflow computation in sufficient detail that the reader may understand the problems involved with building a fast and efficient W-M unit. Key features of dataflow computation are examined: data and control mechanisms; execution model; high-level languages and data structures. Finally, the high-level architecture of a proposed tagged-token dataflow processing element is examined.

1. Fundamental Concepts

1.1 Data Directed Instruction Sequencing

In the dataflow model of computation, the sequence in which instructions are executed is solely dependent upon the availability of data. In other words, an instruction is enabled or "fired" when all of its operands are available. This is in contrast to a typical control flow processor (e.g. 80386) where instructions are executed when "pointed to" by the contents of a centralized program counter.

In a typical computer, the flow of control travels from one instruction to the "next" instruction as directed by the program counter. In a dataflow computer, the flow of control is exactly the path that the data travels. In essence, the arrival of a data token at an instruction is analogous to the conventional program counter pointing to the "next" instruction.

1.2 No Memory Cells

In the dataflow model, there is no concept of storage for variables or of memory cells. The elimination of memory cells prevents the use of shared variables and therefore side effects. This restriction thus keeps the instruction sequencing tied solely to data dependency. For example, if an instruction can receive operands from two sources (the operands are obviously shared variables) how can it possibly know when to fire? The event of firing is crucially dependent on timing, e.g. when is the operand valid for a particular firing of this instruction? However, if operands can only arrive from a single source, the instruction fires whenever the operands arrive. The event of firing is time independent and data dependent.

If the instruction sequencing is tied solely to data and not time dependency, it does not matter in what time order many instructions are executed. The correct answer will be produced by many possible evaluation sequences except for those which do not terminate due to data dependencies. This characteristic is known as the *Church-Rosser property.*

1.3 Language Oriented Architecture

The architecture of a dataflow computer is directed by the language it executes. The dataflow computational mechanism is described with a graphical base language that forms an abstract model of parallel computation. This base language is known as the *dataflow program graph.* It serves both as the machine language to be directly executed by hardware and to define and describe the high-level language used by the programmer. Thus, the architecture is defined by the base language not vice versa.

2. Dataflow Program Graphs

2.1 Acyclic Graphs

A dataflow computer program may be described by a directed graph called a dataflow program graph. The dataflow program graph can be considered as the assembly language of a dataflow computer.

The program graph consists of nodes and arcs. Each node in the graph represents an operation. An operation can be an add, a conditional statement, or even another subgraph. Nodes operate as purely mathematical functions in that their output is solely a function of their inputs, and not of any preexisting state. The rule that determines when an operation may take place is called the "firing rule". This rule states that an instruction may execute, or "fire", when all of its arguments have arrived.

Arcs between nodes represent data dependencies between operations. Tokens, which carry data values within them, flow along these arcs. The arcs are assumed to be first-in, first-out (FIFO) queues of unbounded capacity [Arvind 86]. Result tokens find their way to the next instruction by means of a destination carried within them. The destination is the address of the instruction at the end of the arc. The destination address is necessary because the token actually travels with many other tokens through a large interconnection network. The arc is not a "hardwired" path, but is created "on the fly" by the interconnection network. The destination is used by the network routing logic to steer the token to its destination.

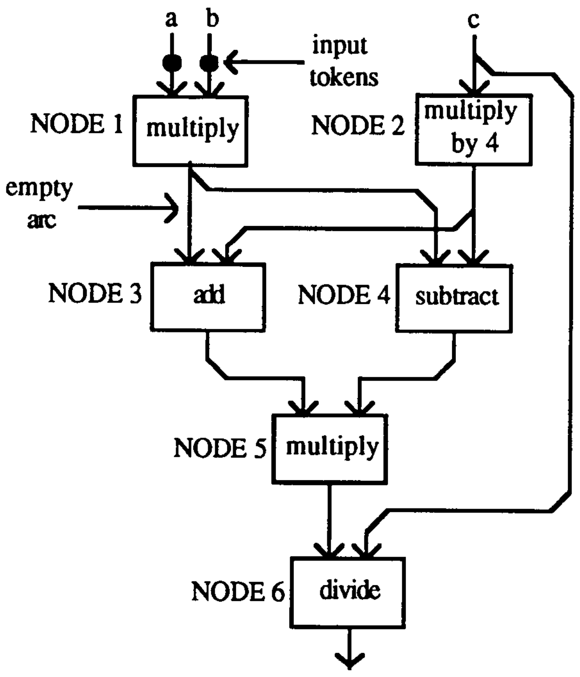


Illustration 1
Program graph for f(a,b,c,d) = [(a*b)+(4*c)]*[(a*b)-(4*c)]/c

In illustration 1, an acyclic dataflow program graph for the function f(a,b,c,d) = [(a*b)+(4*c)]*[(a*b)-(4*c)]/c is shown. Node 1 is enabled for firing while all other nodes are not. Once node one fires, as shown in illustration 2, the tokens that were present at the node inputs are consumed, the multiply is performed, and a result token is emitted at the output arcs. Nodes three and four will then fire when a result token is received from node 2.

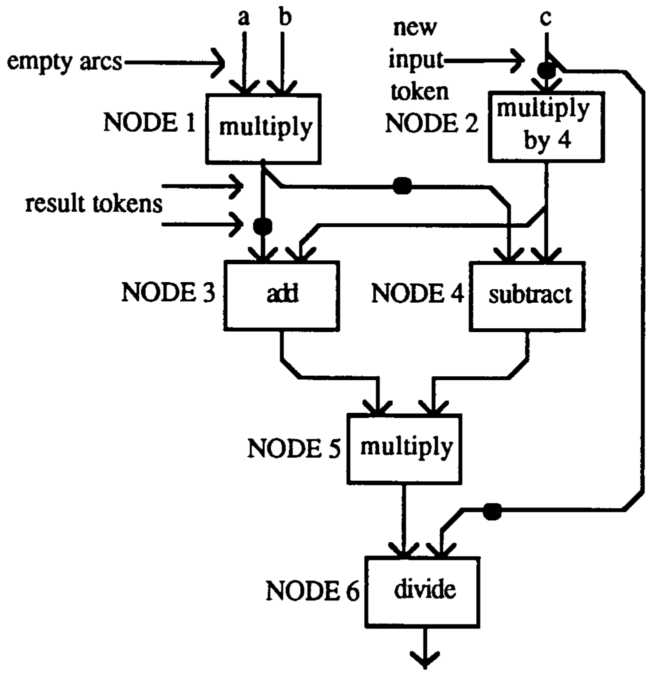


Illustration 2
Program Graph for f(a,b,c,d) = [(a*b)+(4*c)]*[(a*b)-(4*c)]/c
where nodes 1 and 2 have fired

At this point, we notice two important properties of the program graph. First, as long as the firing rule is obeyed, the order of node firing does not effect the output of the graph. Nodes 1 and 2 may fire in any order. Since nodes 3 and 4 cannot fire until tokens from the previous nodes are received, the graph result is the same. This is the property of *determinacy*; i.e. results do not depend on the relative order in which the instructions are executed.

The other property, *parallelism*, is obvious from the program graph. Parallelism is present within the graph when, at some time, there are nodes with no data dependency between them. These nodes may thus execute in parallel [Arvind 86]. In the above illustrations, nodes 3 and 4 have no data dependencies . A further observation is that "waves" of tokens could be applied to the graph allowing evaluation of several waves of tokens at one time. Given the property of determinacy, each wave of tokens will cause emission of a single determinate result.

2.2 Token Storage Methods - Static versus Dynamic Models

An unlimited depth FIFO queue is a convenient conceptual tool, but in reality it can not exist. However, the existence of tokens in the dataflow model implies that some form of storage must be provided [Arvind 86]. How this storage is provided is the main distinguishing feature of the two predominant varieties of dataflow architectures.

The static dataflow model as proposed by Dennis allows only a single data token to be present on an arc at any time [Dennis 80]. This approach thus provides storage for a single token per arc. With single token per arc storage, an instruction must not issue a result token unless the arc for the result token is empty. To implement this token synchronization, handshaking in the form of acknowledgement arcs and tokens is used. Acknowledgement arcs and tokens however, increase token traffic by a factor of 1.5 to 2.0 [Arvind 86]. Their use also increases the time between the firing of adjacent instructions and reduces the amount of parallelism that can be exploited.

The dynamic dataflow model allows multiple tokens to be present on a arc. Token storage is allocated dynamically at run time as needed. While not providing infinite storage, this is a much closer approximation to the basic dataflow model. The arcs in this model do not exhibit FIFO ordering. Instead, tokens are "tagged" with an identifier that indicates their logical ordering within the arc. Thus, large numbers of tokens can arrive in any time order without confusion because logical FIFO ordering is enforced by tagging the tokens.

2.3 Non-Acyclic Graphs - Loops and Functions

As just mentioned, each token in the tagged-token model carries the destination address of the instruction for its data value. However, the destination address forms only one part of a token's tag. The tag also contains an identifier which names the particular iteration of a loop that a data value may belong to. For example, values computed in the i^{th} iteration of a loop are identified by an iteration value in their tags. The full significance of this will be seen shortly in the discussion of loop unfolding.

Within the tag is still another identifier that names the context in which the firing of the instruction is to take place. The context identifier keeps different instantiations of loops or functions distinct. In the case of nested loops, variables from each iteration of the outer loop belong to a new context. Each variable in a particular instantiation of a function call also carries a unique context identifier. Thus, the same program graph may be used simultaneously by arbitrary numbers of contexts. This allows concurrent execution of recursive functions and iterations of loops.

Shown below in illustration 3 is a program graph that computes $\Sigma F(i)$ over i=1 to N. Some new symbols are introduced in this graph. The switch operator routes a token from its input to either its True or False output depending on the value of the control token. Noting that FIFO ordering of tokens is not necessary, we also introduce the non-deterministic or "first-come-first-served" merge. This symbol indicates that two arcs converge at that point. This is allowed because the ordering of the tokens is under the control of the token tags. The operators D and D^{-1} do not operate on a token's data value but rather on the iteration portion of the token tag. The D operator increments the iteration number while the D^{-1} operator resets the iteration number to zero.

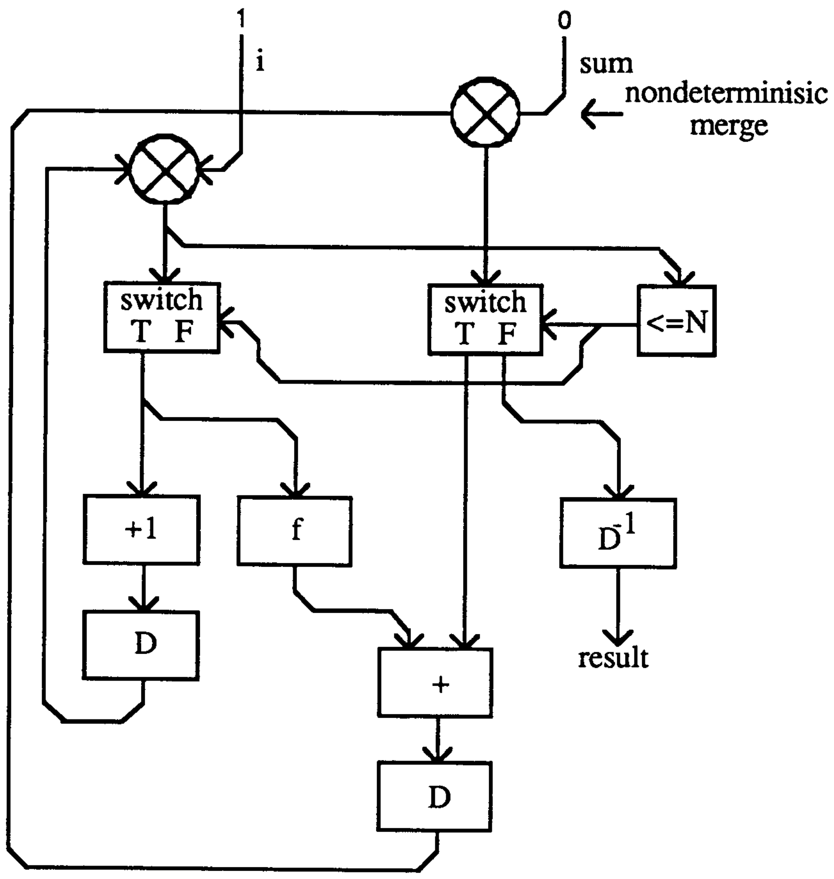


Illustration 3
Program Graph of $\Sigma f(i)$ over i= 1 to N

Initially, the "i" and "sum" tokens are input into the graph. After that, as long as i < N, the values of i and sum will continue to recirculate, passing on the "true" side of the switch operator. When i > N, sum will be output from the false output of the rightmost switch operator and new values of i and sum may be admitted to the loop.

Suppose we define the function f to be an acyclic graph like that in illustration 1. Also assume that function f is very complex and takes a long time to execute. During the loop's execution, the portion of the graph that generates successive values of i may proceed at full speed. Up to N values of i may "stack up" at the input to function f. Each value of i will be tagged with a unique iteration number. Thus, up to N instances of the function f may be started concurrently. Each instance could use the same graph of function f or the instances could be "farmed out" to other processing elements. The other PE's would hold copies of f's graph allowing totally concurrent execution of all instances of the loop.

The generation of separate instantiations of a portion of a program graph is called *loop unraveling* or *loop unfolding*. The tagged-token dataflow model is able to orderly control each concurrent computation under loop unraveling, greatly increasing concurrency. In fact, it has been shown that no other architectural model can exploit more parallelism than the tagged-token approach [Arvind 86].

3. High Level Dataflow Languages

Every programming language is based on a model of a computing system which programs written in that language control. The model may be a pure abstraction or a piece of hardware. High-level dataflow languages (HLDLs) have for their basis the program graph; a naturally parallel semantic base. This is in contrast to conventional programming languages which are based on the von-Neumann physical architecture.

Given the basis of HLDLs, what would the characteristics of such languages be? We would certainly expect the language to have a close correspondence to the dataflow program graph. In particular, the language would not allow side effects. This would naturally allow instruction sequencing to be dependent only on data availability. Furthermore, procedures or function calls would exhibit no history sensitivity due to the lack of any memory in the base language model.

3.1 Functional Programming Languages

Functional or applicative programming languages are those which operate by the application of functions to values. The basic operation in these languages is that of function application. Functional language programs are functions in the true mathematical sense. They are applied to the input of the program; the result is the program's output. The output becomes simply another form of the function and its inputs [Vegdahl 84] [Ackerman 82].

The origin of functional languages is the *lambda calculus*. The lambda calculus was a result of Alonzo Church's work on basic computability theory in the 1930's. [Treleaven 84] Church was attempting to define and express in a simple calculus, the intuitive notion of which functions of positive integers may be computed in a mechanical or algorithmic way.

The lambda calculus is a set of rules governing the conversion of lambda expressions. Evaluation of a lambda expression involves the repeated application of the rules to reduce the expression to its simplest form. The lambda calculus makes a reasonable foundation for a programming language because the reduction of lambda expressions corresponds to evaluating the result of a program. Also, there are several important attributes of languages based on the lambda calculus that make them attractive for dataflow computing.

In reducing lambda expressions, several choices of "what to do next" may exist. The main theorem of the lambda calculus, and one of purely functional languages, the Church-Rosser theorem, states that it does not matter what order things are done in, the result is the same [Treleaven 84]. This guarantees that the order in which subexpressions are evaluated in obtaining a solution is irrelevant. A correct answer will be generated by possibly many evaluation sequences, although some sequences will not produce an answer due to data dependencies.

This key property makes functional languages ideal for dataflow computers. It allows instructions to execute asynchronously, constrained only by data dependencies, and yet despite many possible orderings of instruction execution, the answer will be the same. In other words, a determinate solution is obtained despite non-deterministic instruction scheduling. Thus, many instructions may be available for execution at any given time, allowing exploitation of maximal parallelism within the program.

Another property of purely functional languages is that they exhibit *referential transparency*. This means that the result of an expression depends only on the meaning of its component subexpressions and not on the history of any computation performed prior to the evaluation of that expression. This allows an expression representing a variable to replace occurrences of the variable in a program without changing the meaning of the program [Traub 86]. Referential transparency thus allows easy detection of data dependencies and generation of program graphs from the high level functional language.

3.2 Functional Language Data Structures

The property of referential transparency imposes an important restriction on the generation of results from a function. First, once a variable has been created, it cannot be changed. Thus the appearance or availability of the output from a function must be an all-at-once occurrence. Else, the variable would have changed over time. A result may not incrementally materialize as in the "filling in" of the elements of a data structure.

For example, a row of an array is to have all its elements set to zero in a **for** loop. Because no variable may be updated in place, each time an element is set to zero, a new array must be created. If there are *n* elements in the rows of the array, *n* intermediate arrays will be generated. When any element in a data structure is changed, a new copy of the structure must be made. This means an enormous overhead spent in copying nearly identical data structures.

Along with the advantages functional languages provide in a dataflow context, comes the awkwardness of handling structured data. However, there are ways in which structured data may be efficiently manipulated without sacrificing determinacy or parallelism. The solutions sometimes "bend the rules" of functional computation, but the "kinks" are invisible to the programmer.

4. Dataflow Hardware Architecture

The preceding discussion has been quite generic. To further understand the tagged-token architecture, we will now take a look at the MIT Tagged-Token Dataflow Architecture (TTDA). The TTDA, as its name suggests, is a dynamic tagged-token dataflow computer. This particular machine has been chosen as a vehicle for study because of the wealth of available information. Also, other tagged-token machines that have been or are being designed are very similar to this particular architecture.

4.1 TTDA Basic Configuration

The TTDA is a multiprocessor that consists of a number of identical processing elements or PEs which directly execute dataflow program graphs. Each PE constitutes a complete dataflow computer. Within each PE are two main sections, the computing section and the I-structure section. The computing section is where the actual computation takes place. The I-structure section is where structured data items are stored. The I-structure greatly improves structured data manipulation efficiency when functional languages are used [Arvind 80]. A simplified view of the TTDA's structure is given below in illustration 4.

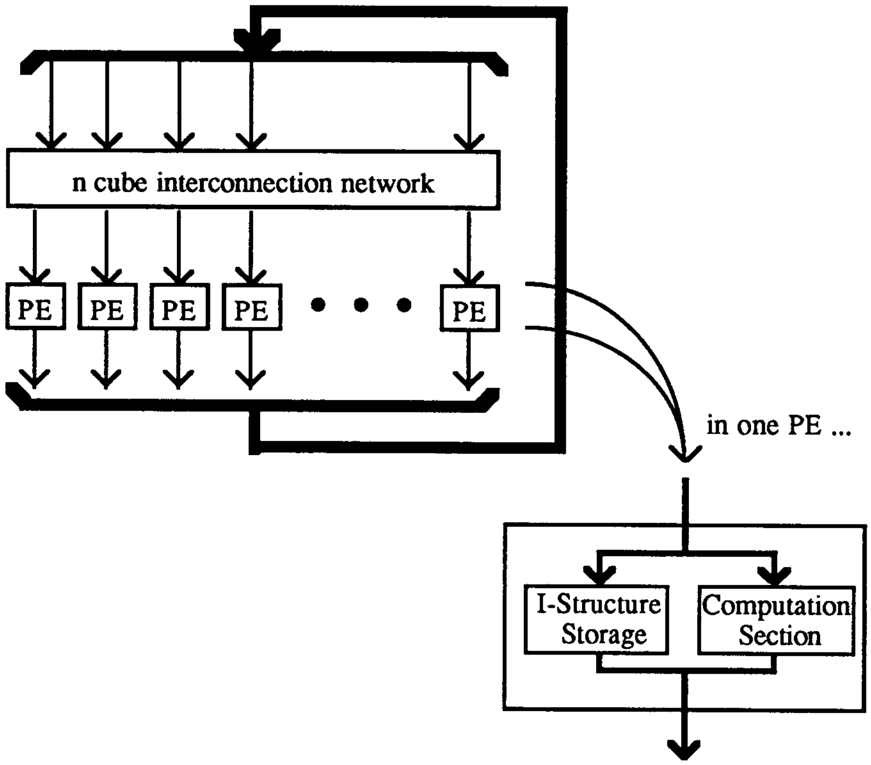


Illustration 4
Simplified View of the MIT Tagged-Token Dataflow Architecture

4.2 Program Representation

Dataflow programs, especially those written in MIT's dataflow language *Id* [1], are partitioned into collections of program graphs known as *code blocks* [Nikhil 86] [Traub 86]. A code block typically consists of a compiled function, nested function or a loop. Code blocks form the level of granularity for the dynamic distribution of work among PEs. It should be stressed that the distribution of work among PEs is not done on the basis of individual instructions. Work within a code block (instruction granularity) is distributed at run time within a PE with hardware support.

[1]The name Id comes from Irvine Dataflow. Dr. Arvind's research first began at the University of California at Irvine.

An instruction in the TTDA can be viewed as in illustration 5 [Arvind 87b]. The opcode field is similar to those for typical ALUs. The literal/constant field holds either a literal value to be used in the instruction or an offset into a constant storage area.

opcode
literal/constant
destination 1
destination 2
• • •
• • •
destination n

Illustration 5
TTDA Instruction Format

The program graph for a code block is stored in the PE's program memory. The description of the graph is formed by the information stored in the destination fields of each instruction. These fields simply hold the address of the following instruction in the program graph. Once an instruction fires, its destination fields are used to build the output token's destination instruction address. There is one destination field for each input to which the output of the instruction is connected. The destination field addresses given are relative to a pointer which points to the beginning of program storage memory for a particular code block.

4.3 Data Token Tag Representation

Shown below in illustration 6 is a simplified diagram of a TTDA data token, showing the various parts [Traub 86].

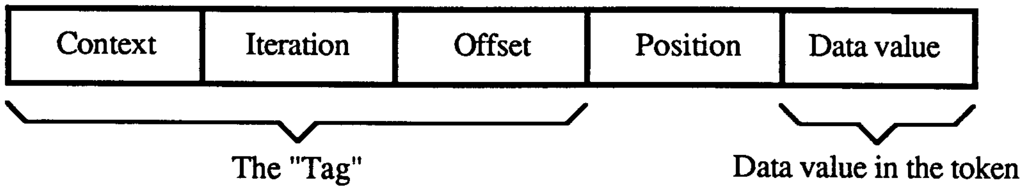


Illustration 6
Format of TTDA Data Token

The context field distinguishes tokens which may be sharing the same code block but belong to a different invocation of that code block. An example would be where multiple invocations of a recursive function are running concurrently. Each invocation of the function would use instructions from the same code block, but tokens from different invocations would be distinguishable by means of their contexts.

The context field also serves as a pointer to three registers. One register, the *Code Block Register*, holds the base address of the code block's instructions in the program memory. Thus, when the tokens have been matched, the address of the instruction to be executed is formed by adding the contents of the code block register and the offset field. Another register the context identifies is the *Data Base Register.* This register points to a base address in constant memory. Using this base address and the constant offset field in the instruction, a constant value may be fetched. The ability to not have circulating constant tokens in the dataflow graph is a simple but very effective improvement [Arvind 86].

The third register indicated by the context field is the *Map* register. It is used to specify mapping algorithms for the distribution of work among PEs. It will be discussed later.

The iteration field indicates to which iteration of a loop a token belongs to. It's meaning is similar to the context field. It distinguishes between tokens which are sharing the same instructions within a loop code block but that belong to a different iteration of that loop.

In the case of nested loops, the context and iteration fields are used together to maintain a token's identity. The outer loop generates *n* invocations of its loop. Each of these loop invocations will generate *m* invocations of the inner loop. To distinguish tokens that are within the inner loop, each new invocation of the inner loop is given a new context. If the outer loop has *n* loops. then *n* new invocations of the inner loop will be generated each with a unique context. Each invocation of the inner loop is then free to unravel its loops.

The offset field, as mentioned previously, is used to indicate the address of the instruction to be executed. Its value is added to the code block register to obtain the instruction address.

The position field indicates to which input of the instruction the token is destined. TTDA instructions limited to receiving one or two tokens for firing. Tokens requiring only one token for firing are identified by an unshown bit known as the *Partner-P* bit [Traub 86]. A token with this bit set will bypass the waiting-matching section of the PE. Instructions are limited to two operands for engineering reasons. By having only two tags to match, associative or pseudo-associative matching by hashing is possible.

4.4 Processing Element Section

Below in illustration 7 is shown a block diagram of the TTDA PE structure. We will now discuss the function of each major block.

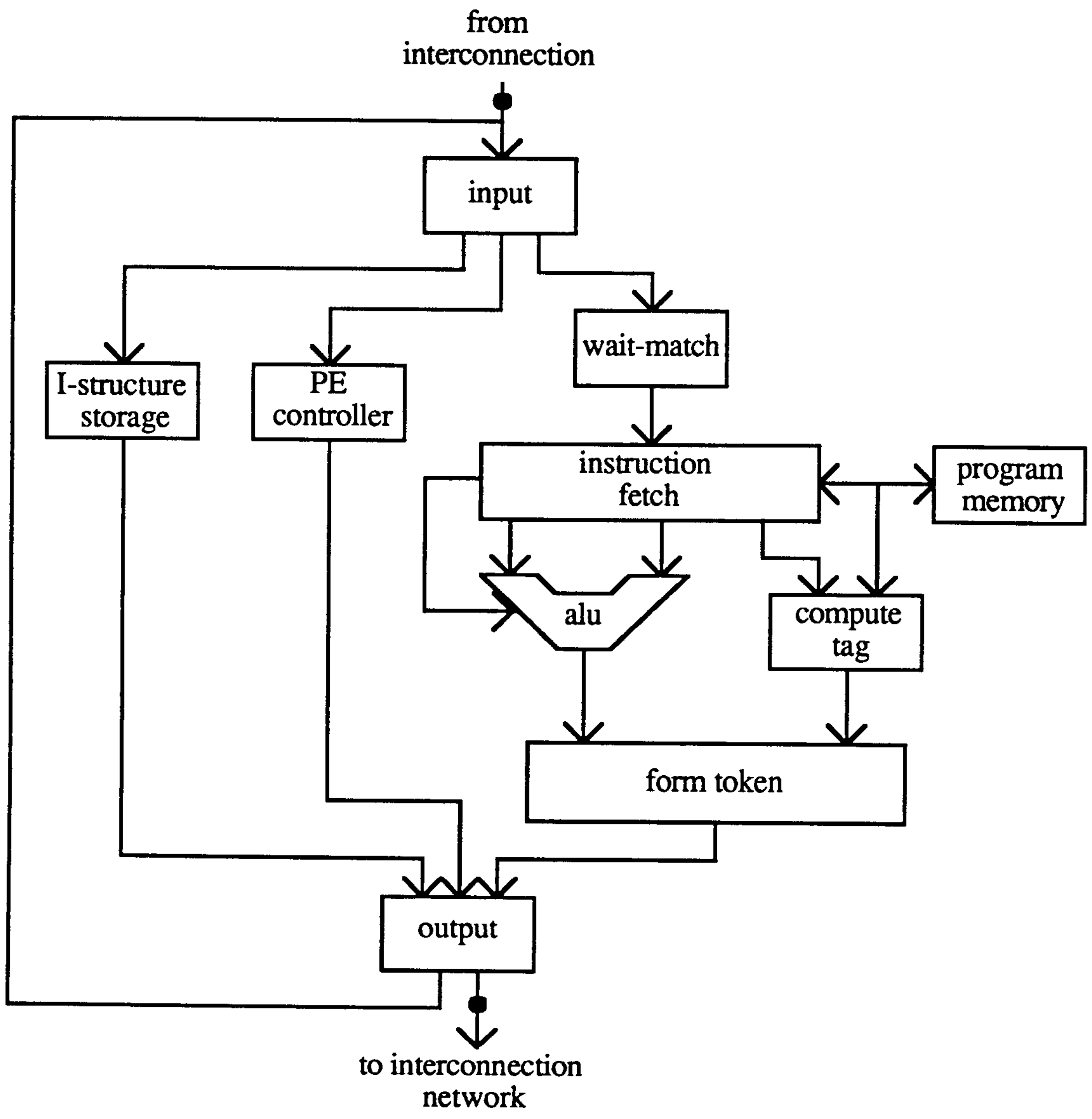


Illustration 7
Detailed Diagram of the TTDA PE Structure

4.4.1 Wait-Match Unit

The Wait-Match Unit (W-M) is an associative memory and controller which matches tokens with identical tags or stores tokens whose mate has not yet arrived. If a token arrives for a monadic instruction, the Partner-P bit causes the token to bypass the W-M unit. When a token arrives destined for a diadic operator, a search is made for all stored tokens in the W-M. If a token with a matching tag is found, both the tokens are passed on to the Instruction Fetch unit. If no matching token is found, the token is stored in the W-M's memory.

4.4.2 Instruction-Fetch Unit

When the matched diadic tokens or single monadic tokens reach the instruction-fetch unit, it fetches the instruction addressed by the sum of the contents of the code block register pointed to by the context field and tag's offset field. At this time, any constants specified by the instruction are fetched from the constant memory using the sum of data base register and the constant offset field in the instruction. The token's data values, any constant values or literals, the opcode, context and destination instruction addresses are then passed to the ALU and Compute-Tag unit.

4.4.3 ALU and Compute Tag Unit

The ALU and Compute-Tag unit consists of two parts. The ALU portion of this unit simply takes the token data values and any constants or literals and computes the result. The Compute-Tag portion uses the context from the token tags with the destination instruction addresses in the instruction and computes a new tag for the result token. It is in this section that tokens may receive new contexts or iteration fields. Typically though, two instructions will lie within the same context and iteration namespace, thus requiring only a new destination instruction address which is supplied by the instruction just fetched. The results from this unit is then passed to the Form-Tokens unit.

4.4.4 Form-Tokens Unit

This unit takes the result from the ALU section and attaches the tag(s) sent from the Compute-Tag unit. The complete result token(s) is then sent to the interconnection network.

4.4.5 Control Unit

The control section receives special control tokens which can manipulate the state of the PE. These tokens are typically generated by various operating system resource managers. The control section also controls connections to the outside would such as input-output devices.

4.4.6 I-structure Unit

Most tokens destined for a PE follow the route to theW-M unit. Tokens may also be destined for the I-structure unit. The I-structure unit provides a storage and manipulation facility for structured data items such as arrays or records. Locations within the I-structure are read in a two-stage process. First, a special I-structure read token is sent to the I-structure controller requesting the read. This token contains the I-structure address to be fetched from as well as the context, iteration, offset, and position for the return data to go to. If the location has been written to, the I-structure unit responds with a token which contains a tag identical to the read request token, and the data item fetched from the structure. If the location to which the request had been made has not been written to, the read request is deferred. The request is queued on a deferred-read list for that location. When the location is finally written, the stored value is sent to every requestor queued on the deferred-read list [Arvind 80].

Storing a data item into the I-structure is done with a special I-structure store token. This token holds the address in the I-structure and the data value to be stored. If an attempt is made to write to a location which already holds a data item, an error is generated. This is an illegal I-structure operation. The updating of the location would violate the single assignment principles of dataflow computation.

It should be noted that if a PE sends a read token to the I-structure unit, it does not have to wait for the response token to arrive before continuing processing. The ALU is free to execute any other enabled instruction once the I-structure fetch token is sent. While the response token is yet forthcoming, the PE can do instruction level context switching, executing enabled instructions. This instruction level context switch is accomplished without any "state saving" overhead. This action enables the dataflow PE to tolerate memory latency.

III. Waiting-Matching Unit Requirements

1. Introduction

As previously stated, the power of the dynamic dataflow model is its ability to exploit all the parallelism existent within a program. This is possible for two reasons. First, the semantic base of the language is able to expose all of the parallelism to the hardware. Secondly, the hardware is able to support many concurrent computational activities by explicitly tagging data tokens with the context for which they belong. This allows for example, all the different loop iterations in a **for** loop to execute concurrently from the same code block. The context unique tags allow data tokens from concurrently executing loops to be matched without confusion.

The operation of matching or storing tokens conditionally depending on their context is the job of the Waiting-Matching (W-M) unit. When an incoming token arrives at a PE's W-M unit, the tag or context portion of the token is associatively compared with all the previously unmatched tokens stored in the W-M unit. If the token's tag matches the tag of another stored token, the stored token is extracted from storage and the operand portions of both matching tokens are delivered to the ALU. Then the previously used storage location is garbage collected. If a match did not occur, the incoming token is stored in the W-M unit.

At a high level, the operation of the W-M unit is easy to understand. However the difficulty of implementing such a unit in hardware is formidable. The structures required to perform associative comparisons against large numbers of stored items do not map well to existing semiconductor devices and technologies. To date, only very small content addressable memories (CAMs) have been built, most as microprocessor Translation Lookaside Buffers (TLBs) which typically hold 64 or fewer entries. In contrast, the token storage requirements for small dataflow computers have been estimated to be in the tens of thousands of tokens per processing element.

Since the W-M unit's output forms at least one input to the ALU, its speed should be at least equal to the cycle time of the ALU. The W-M unit essentially acts like a register file would in a conventional computer. But conventional computers rarely have more than 32 registers to keep their access time up to that of the ALU. Clearly, there is a conflict between the speed and size requirements of the W-M unit.

In this section, we will focus attention on the requirements of the Waiting-Matching unit. The required speed and size of the W-M unit are quantified and a description of an ideal W-M unit is given. The section concludes with a brief discussion of balancing ALU utilization with matched but queued tokens.

2. Waiting-Matching Unit Size

Even in programs where no effort has been made to use parallel algorithms, dynamic dataflow techniques can expose prodigious amounts of parallelism [Culler 87]. In some highly parallel programs, the amount of parallelism exposed can actually be excessive and must be limited.

Whenever new contexts are spawned, tokens on each arc of the program graph must have storage available. Tokens stored in the waiting-matching unit are removed only when matched to an incoming token. Thus, if the waiting-matching unit becomes completely filled with unmatched tokens, the machine will immediately deadlock [Arvind 86]. For Arvind's group, this was a serious enough problem that for some time they generated only those dataflow program graphs in which parallelism was bounded [Arvind 86].

Arvind's group has recently been investigating the token storage requirements of their TTDA. Using an ideal execution model they have determined the token storage requirements for sample codes. Their model assumes a dataflow computer with unbounded processors, unbounded storage resources, unit time operation and zero communication latency. Two metrics were devised to describe the amount of parallelism present in a program and its storage requirements. These metrics are the *Parallelism Profile*, the number of concurrent operations over time; and the *Token Storage Profile*, the number of tokens in existence during each execution step [Culler 87].

Shown below in illustrations 8 and 9 is the parallelism and token storage profile for eight iterations of a program known as SIMPLE. SIMPLE is a hydrodynamics and heat conduction simulation program developed at Lawerence Livermore National Laboratory to provide a benchmark for evaluating high-performance computers.

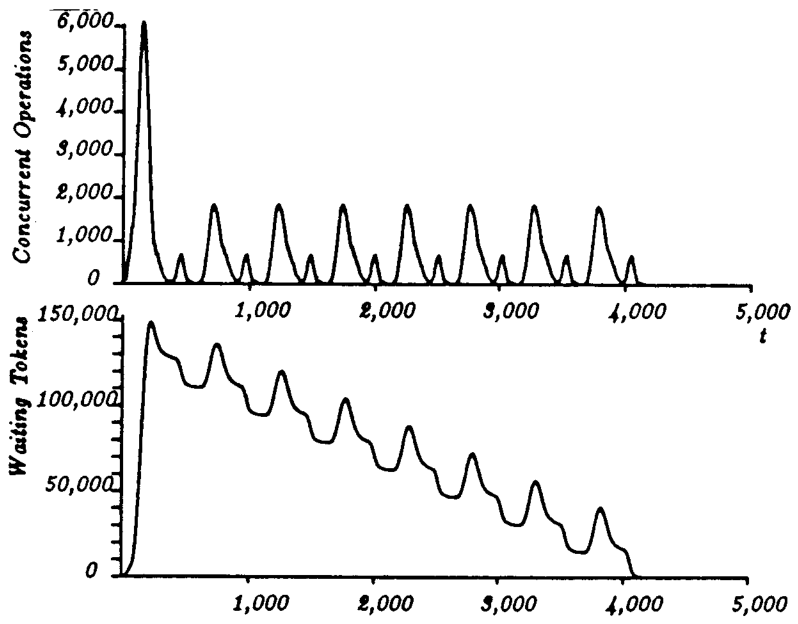


Illustration 8
Parallelism and Token Profiles of SIMPLE with Unlimited Loop Unfolding
[Culler 87]

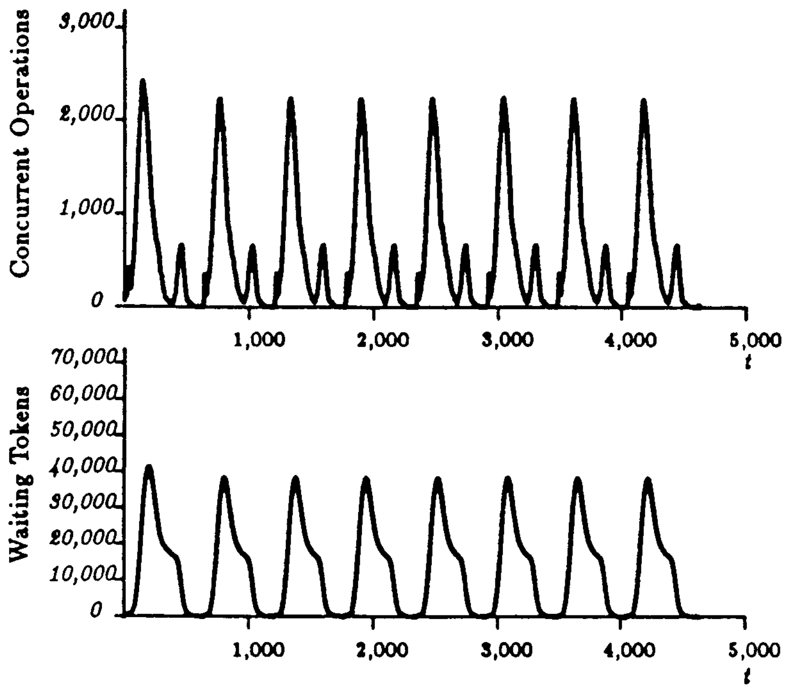


Illustration 9
Parallelism and Token Profiles of SIMPLE with Outer Loop Bound to One [Culler 87]

The eight iterations of the program's outer loop can be clearly seen. Each loop consists of two phases; a hydrodynamics calculation followed by a heat conduction calculation. Each phase computes density, pressure, velocity components and other values in a mesh of zones as time progresses in discrete steps. Each time step may vary depending on the results of the preceding step [Ekanadham 87]. Because of this, values computed in a phase are dependent on the previously calculated time step. This results in the constriction of concurrent activities between each iteration.

Each graph shows curves for two values of outer loop unfolding. Where unfolding was unbounded, all eight iterations of the outer loop unfold immediately. Also, each of the outer loop's inner loops unfold. However, since the computations in each iteration depend on the results of the previous iteration, tokens are spawned in the outermost loops only to wait long periods for their mates to arrive. The exposed parallelism is not useful and serves only to drive up token storage requirements.

By bounding the outer loop unfolding to one, the storage requirements are reduced by more than 65% while increasing total runtime only slightly. With bounded the loop unfolding, the token storage requirements become independent of the number of iterations. Only the size of the mesh determines the storage requirements.

It should be noted however that this is a very small example of SIMPLE. Real problems similar to SIMPLE would involve 100,000 iterations on a 100x100 mesh. By extrapolating their measurements of token storage profiles, Arvind's group have estimated that for a 16 processor system, the token storage requirements for running SIMPLE on a 100 x 100 mesh would be approximately 30 to 100 thousand tokens at each processor.

3. Waiting-Matching Store Speed

Research has shown that tagged-token machine performance is crucially dependent upon the rate at which the W-M section can process tokens[Arvind 86]. This point was made clear by a number of tests run on the Manchester dataflow computer. It was found that when running a program that used only monadic operators which always bypass the W-M unit, speedup increased linearly as the number of processing elements was increased until program parallelism began to limit speedup. See illustration 10. However, if the program used dyadic operators, the waiting-matching section caused pipeline starvation resulting in a much lower performance [Lerner 84][Gurd 85].

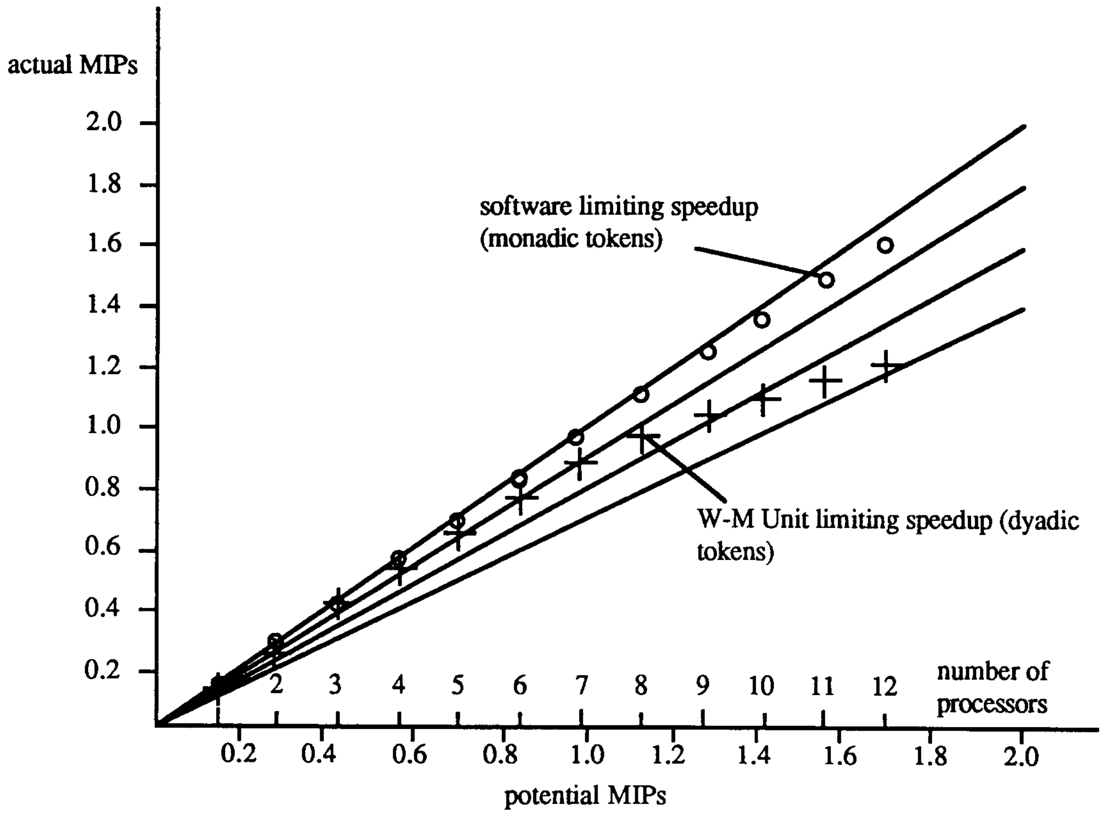


Illustration 10

Manchester Dataflow Computer Performance Curves [Lerner 84][Gurd 85]

To prevent starvation of the ALU, the waiting-matching unit must be able to supply data operands at the rate of up to twice that of ALU output bandwidth [Papadopoulos 88]. This is because the ALU typically consumes two input values to compute one output value. Given that modern ALUs can perform simple operations in less than 20 nS, this places an extreme performance requirement on the waiting-matching section. Fortunately, fifty to sixty percent of all instructions are monadic and do not require matched tokens [Gurd 85]. These monadic instructions provide some "breathing room" for matching unit performance.

4. Target Description for a Real World W-M Unit

Before going further into specific implementations of a W-M unit, it is necessary to list the characteristics of a reasonable, useful, W-M Unit.

4.1 Speed

The speed of the waiting-matching unit should be matched to that of the ALU. If properly matched, the W-M unit must not cause tokens to get "backed up" into the interconnection network or allow the ALU to become starved for matched token pairs. Within a single ALU cycle, the W-M unit must :

1. compare the incoming token with all tokens in the store;
2. if a match occurs, release the data portion of the stored token and garbage collect the previously used location; else, store the incoming token.

A reasonable assumption is that the ALU in a VLSI dataflow processor would have a single-cycle execution time of about 20nS. Since the ALU will typically process two operands to produce a result, the W-M unit should be able to supply matched tokens at twice the rate of ALU consumption or about 100 million tokens per second. The two phase "match-garbage collect" operation of the W-M unit implies the use of two W-M units to keep the ALU properly supplied with operands if each phase of the operation also takes 20nS. This approach is somewhat conservative, considering that half of the incoming tokens may be monadic.

4.2 Size

There must be enough token storage available at each PE to satisfy the needs of the application that is running. Estimates for this storage range from 10,000 to 100,000 tokens per PE. A conservative token storage capacity of 128,000 tokens will be chosen. However, regardless of the token store size, at some point some operator error will cause an inadvertent overflow of the token store causing the computer to deadlock. Some method of gracefully recovering from or avoiding this unacceptable condition is needed. This issue is not addressed in the paper.

4.3 Level of Integration

The W-M unit must be placed close to the ALU to provide single-cycle throughput. This is because of the speed penalty incurred whenever signals leave the confines of a chip. Thus the W-M unit must be small enough to fit on the same silicon die with the ALU.

5. Token Storage or Wasted Parallelism?

There is an important point to keep in mind while considering token storage within a dataflow machine. According to Arvind, to exploit parallelism in algorithms, any high performance multiprocessor system must be able to tolerate long latencies for memory requests [Arvind 83][Arvind 87a]. The dataflow computer can do this by decoupling memory latency from instruction execution. This is done by rapidly context switching at the instruction level by having a queue of paired tokens always available. This keeps the PE's continually busy, never waiting on memory responses. The point is that queues of paired data tokens represent parallelism that is being absorbed or wasted instead of exploited.

What we want is 100% ALU utilization with minimal absorption of parallelism. This would occur where a set of matched tokens with their instruction are made available to the ALU just as the last instruction was completed and the result token is emitted. This of course, may be extremely difficult to do in practice. It would require predicting the latency of remote memory requests, knowing the amount of single token instructions to run, predicting the amount of parallelism in the program, and other parameters that are difficult to ascertain a priori.

Compiler techniques could help however by proper scheduling of instruction execution. A compiler could "time" the arrival of tokens to a W-M unit having them arrive at the same time to minimize the number of stored and unmatched tokens.

IV. A Hierarchically Structured W-M Unit

1. Memory Hierarchies

In conventional and dataflow memory systems, at fixed cost and technology, the speed of a memory system is inversely proportional to its size. The desire however, is to have a big memory that is as fast as a small memory. To achieve this, conventional computers use a hierarchical memory structure.

In this hierarchy, the fastest memory is located closest to the ALU. In conventional computers this memory consists of the general purpose registers, where the most often used data items are kept. The next level of memory is slightly slower and larger and will hold the not-quite-as-recently used data items. An off-chip cache memory might represent the second level memory. The next level of memory is even slower and larger and will hold infrequently used data items. The main DRAM memory would be a typical third level memory. The bottom layer is much slower and larger bulk storage such as magnetic disk that holds very infrequently (relatively speaking) used data items.

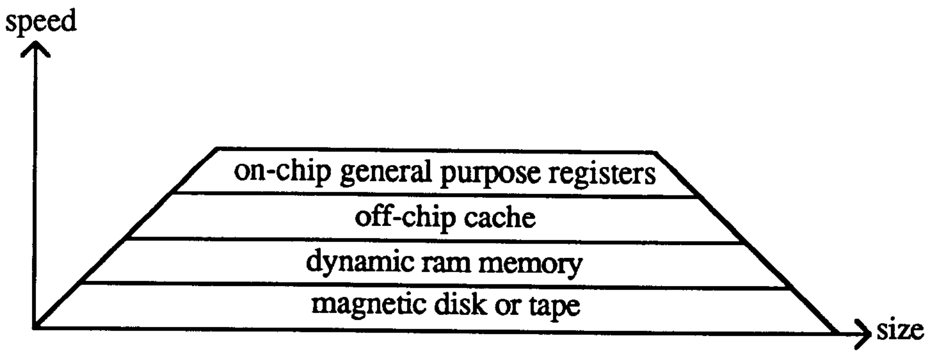


Illustration 11
Conventional Memory Hierarchy

If a memory hierarchy is properly designed, the average access time for the memory system is close to that of the fastest memory. This is because in conventional addresses that were referenced recently to be accessed again in the near future. This is temporal locality. There is also the tendency to reference addresses near to the one last referenced. This is spatial locality. Also, addresses to be referenced next are often the one after the last address referenced. This is sequential locality. By keeping the most often referenced data items in the fastest memory, the memory hierarchy takes advantage of these three types of referential locality. The average access time for memory references in a hierarchically organized memory can be represented by [Hwang 84]:

$$Tacc = \sum_{i=1}^{n} (h_i * Tacc_i)$$

Where, Tacc is the average access time,
h_i is the hit rate at the ith level of the memory hierarchy
$Tacc_i$ is the access time at the ith level of the hierarchy

2. Referential Locality in a Dataflow Computer

In conventional computers, high memory bandwidth is realized by keeping the most often referenced data at the fastest level of the memory hierarchy. However, dataflow computers do not reference memory locations like conventional computers and thus have no memory referencing pattern. In a W-M unit, tokens are written into unspecified locations once and read only when a match occurs, and are then discarded. Thus a hierarchical structure or caching strategy based on spatial or sequential locality would be useless. Fortunately, dataflow computers do have a token referencing pattern which possesses a strong degree of temporal locality.

Tokens circulating in a dataflow pipeline tend to fall into one of two sets distinguished by their temporal locality. In the first set are tokens used as intermediate values in an active code block. These correspond to values held in the local registers of a conventional computer. The second set of tokens hold the context for suspended procedures. These data items belong to presently inactive code blocks that are waiting for a value or values to be returned from a procedure call from within its body. These tokens are analogous to register variables that have been pushed on the stack upon activation of a procedure call [Brobst 87].

Time partitioning of token lifetimes suggests that a hierarchically organized W-M unit may be useful. A small CAM could used to match tokens from active code blocks and also act as a fast temporary storage for tokens awaiting their mates from active code blocks. This fast first-level *token cache* is then supplemented with a slower and much larger second-level waiting-matching unit.

For the token cache to work, we need to maximize the number of tokens in the cache that belong to active code blocks. However, it is not possible to tell upon arrival to which set the token belongs. Fortunately, the temporal locality existing between tokens in the active code blocks allows a grouping by age. See illustration 12 below.

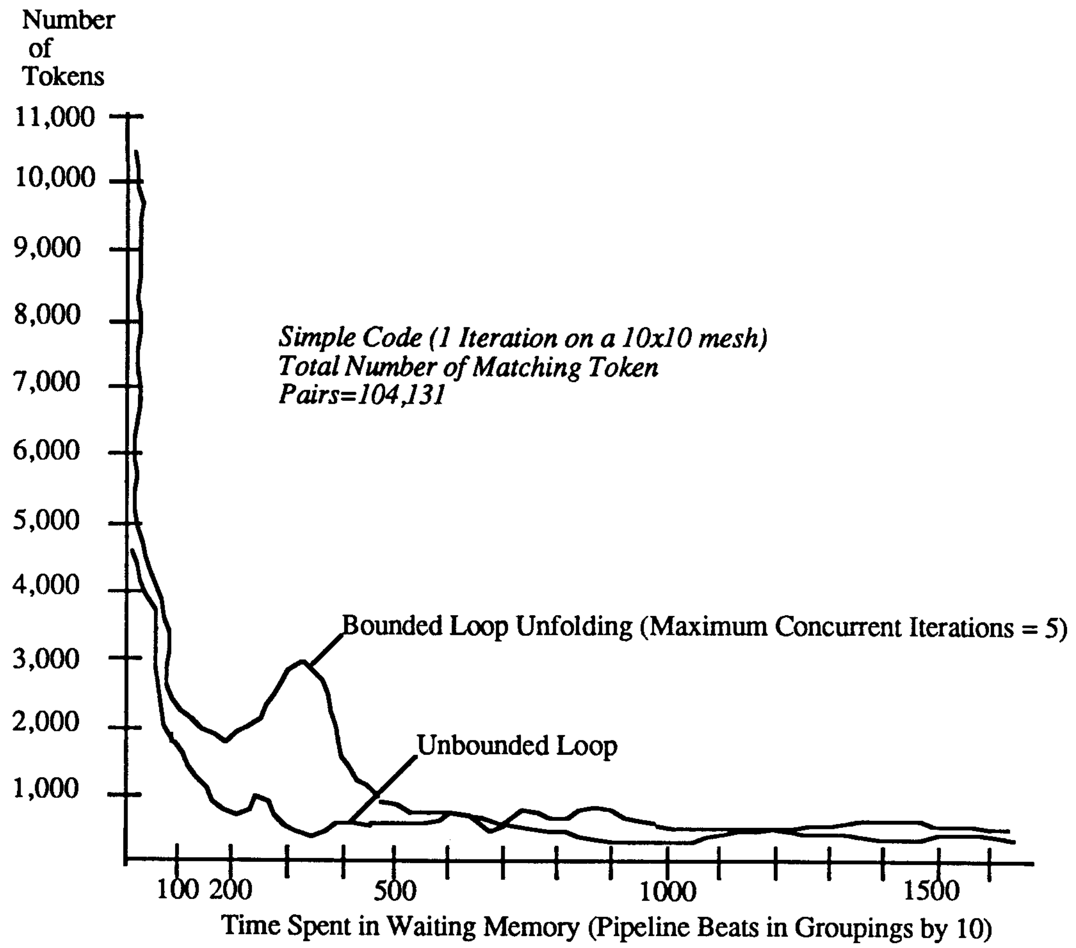


Illustration 12
Token Temporal Locality [Brobst 87]

Note that the majority of tokens spend less than 400 pipeline beats in the W-M unit before being matched and purged. These tokens belong to code blocks that are currently active. Thus, when a token arrives, its partner is likely to arrive within a four hundred pipeline cycles given the constraint of bounded loop unfolding. If it doesn't, it is probably a token from an inactive code block. This behavior suggests a simple FIFO replacement policy for the cache. If a token's partner has not arrived within a given waiting period it is removed from the cache and placed into the secondary level waiting-matching unit.

The degree of loop unfolding has a large influence upon the average token lifetime. When loop unfolding is limited to five, an overwhelming majority of tokens exist in the W-M unit for less than 400 pipeline beats. When loop unfolding is unlimited, temporal locality is greatly diminished. This indicates that a careful match must be made between token cache size and the amount of loop unfolding allowed. To allow for more unfolding, a larger cache must be present.

3. Design of the Hierarchical W-M Unit

In this section we will look at the design of a hierarchical W-M unit. Emphasis will be placed on the "top" or primary level of the hierarchy. This approach is justified because the token cache hit rate is very high making the performance of the secondary level matching unit relatively insignificant.

3.1 High Level Organization and Operation

Shown below in illustration 13 is the block diagram of the MIT TTDA with its W-M unit in bold. The inset drawing shows the high level view of the hierarchical W-M unit. It consists of a primary and secondary waiting-matching unit. This drawing shows how the hierarchical W-M unit could be incorporated into a dataflow computer such as the TTDA.

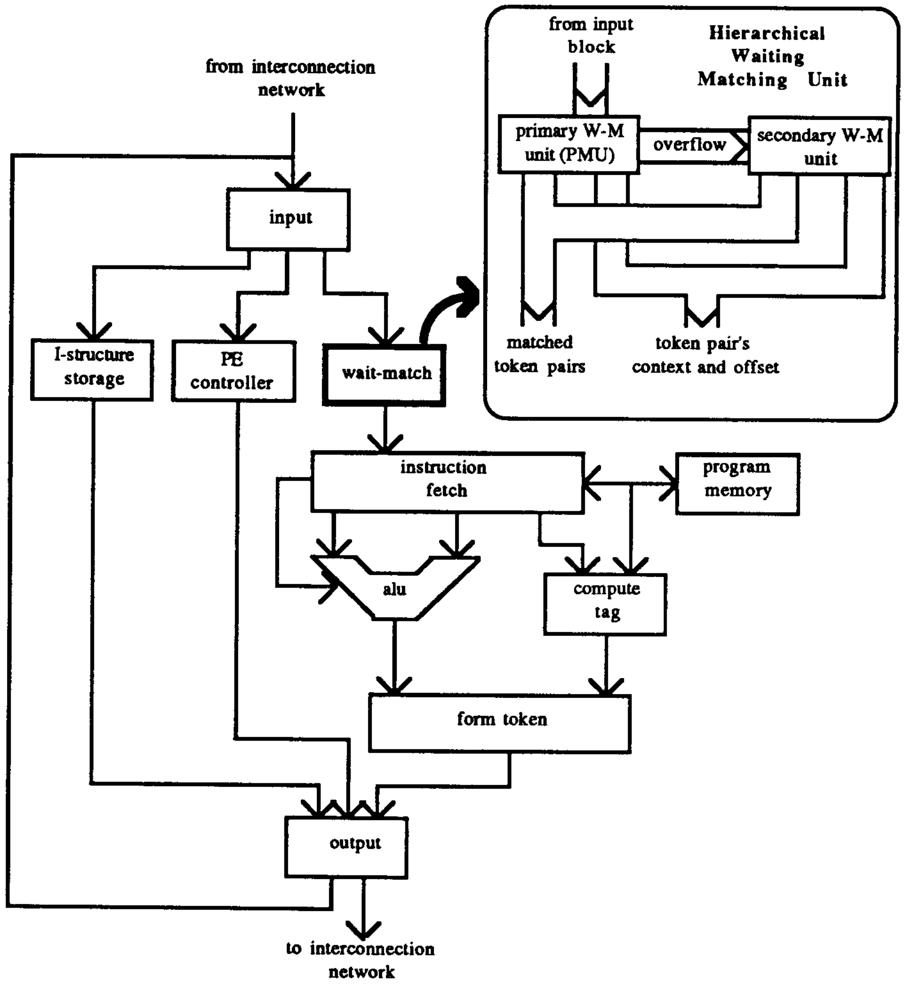


Illustration 13
MIT TTDA with Inset of Hierarchical W-M Unit

3.1.1 Waiting and Matching

Tokens from the input block first enter the primary matching unit (PMU). The PMU acts as the token cache. It holds the 256 most recently arrived tokens. If an incoming token finds its mate in the PMU, the data portion of the incoming and the stored token are passed to the ALU. The matching context and offset for the token pair are passed to the instruction fetch unit. The instruction fetch unit then uses the context and offset to access the destination instruction and any constants. (See Ch.II, sec, 5.3, 5.4).

If the incoming token does not find its mate, it is stored in the PMU. Subsequent tokens not matching any other stored token are also stored in the PMU in FIFO order, "on top of" the unmatched tokens. Tokens are effectively aged by the FIFO ordering. This process continues until either an incoming token matches the "lonely" token or until incoming tokens eventually "retire" the unmatched token to the secondary W-M unit. Matching of old tokens also occurs in the secondary W-M unit. The results of such a match are forwarded to the instruction fetch unit as from the PMU.

3.1.2 Miss Policy

The hierarchical W-M unit is unusual in how it handles misses to the token cache. In a conventional computer, a cache miss initiates an access to the next level of the memory hierarchy to fetch the desired item. For a dataflow computer, this does not make sense. Fifty percent of the tokens coming in are guaranteed not to find their partner in the primary or secondary storage. If the secondary storage were accessed on each miss the average access time of the memory system could approach that of the secondary store. The best policy is to leave the unmatched tokens in the primary storage in anticipation of the arrival of their mate.

Some small percentage of token pairs will have one token in the token cache and the other in the secondary matching unit. It would be advantageous to be able to match tokens across the boundary between the PMU and SMU instead of waiting for the tokens in the PMU to migrate by aging. A possible way to do this would be to use idle PMU and SMU cycles to allow the secondary store to check the tokens in the primary store for mates. If the token in the primary memory does not match it is marked as checked, and the search continues.

Another idea comes from remembering that many tokens that have "aged" into the SMU correspond to local variables from suspended procedures that have been pushed onto a stack frame in a conventional computer. Thus, the arrival of a token that matches a token in the SMU probably indicates the reactivation of the suspended procedure. Tokens identified by tags with similar context and iteration numbers would be brought back into the primary matching unit for fast matching of the remaining tokens from the reawakened procedure. This "prefetching" of tokens is an attempt to exploit code block locality. This idea is not explored further in this thesis.

3.2 Structure and Operation of the PMU

We will now concentrate on the organization and operation of the PMU. The PMU is at the heart of the hierarchical W-M unit and is our major focus. Illustration 14 shows the block diagram of a PMU and how it is connected to the data flow computer.

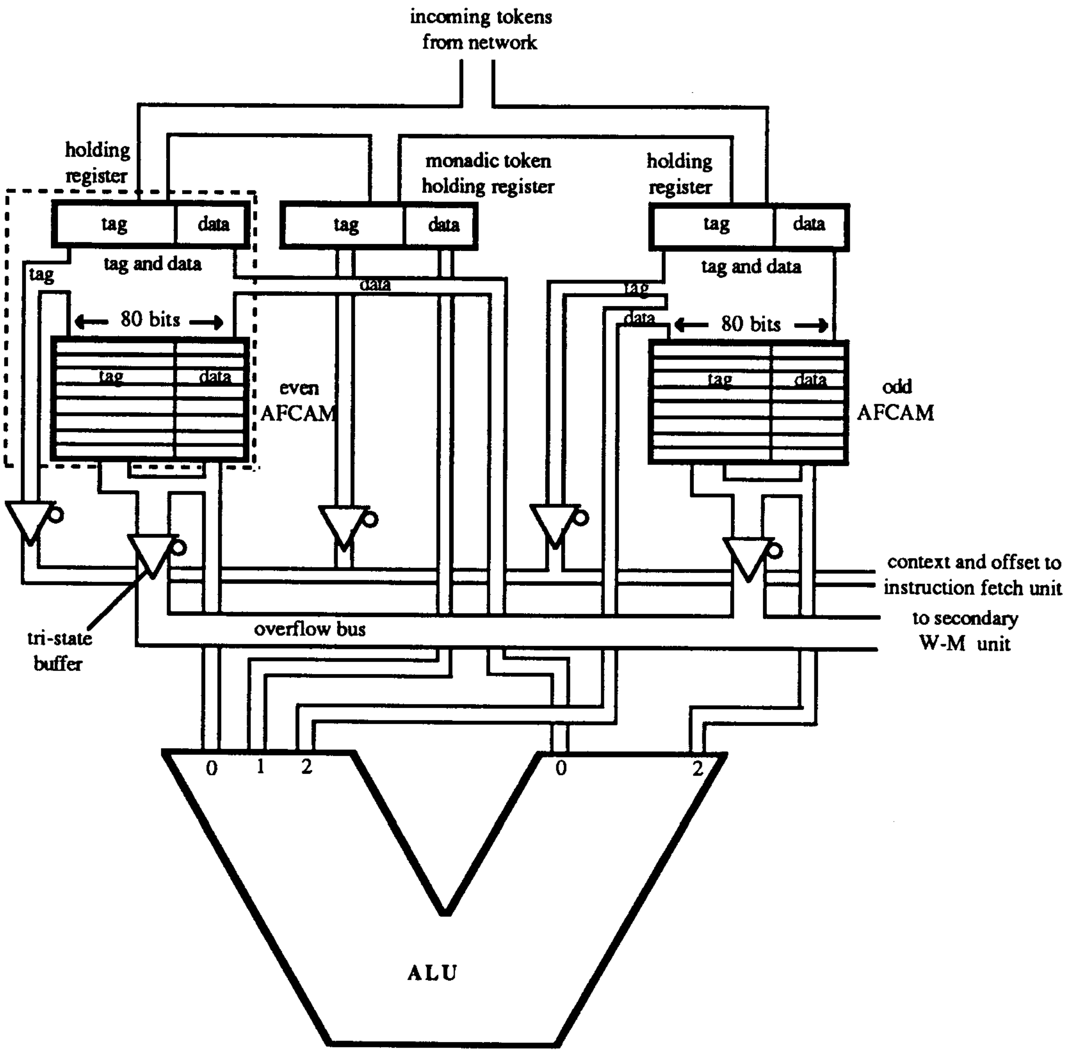


Illustration 14

Close Up of PMU

The PMU consists of two specially structured CAMs called AFCAMs. AFCAM stands for Articulated FIFO CAM. The structure of an AFCAM is shown above inside the dotted area. The AFCAM stores data in FIFO order and also performs associative matching between its contents and a holding register. When matching occurs, the AFCAM "garbage collects" the matching data's memory location in such a way to preserve the FIFO ordering.

Each AFCAM is capable of holding 128, 80-bit tokens, (32-bit data, 48-bit tag) and can provide a matched token pair every other clock cycle. To provide single-cycle delivery of matched tokens to the ALU, two AFCAMs are used. Incoming tokens can be diverted to even or odd AFCAMs depending upon the least significant bit of the token's destination instruction address field. This would provide a fair and even distribution of tokens to each AFCAM.

At the input to each AFCAM is a register that holds the token that is presently looking for its partner in that AFCAM. If the token's mate is not found, the token is stored in the AFCAM in FIFO fashion. If a match does occur, the portion of the register holding the context and offset fields is output to the instruction fetch unit to begin fetching the instruction for the token pair. The data field in the holding register is applied to one ALU input while the data from the matched token in the AFCAM is applied to the other input. Should either AFCAM begin to overflow, the oldest token is transferred to the SMU.

Tokens which are monadic bypass the PMUs and are registered at the monadic token holding register. The context and offset in the register are then output to the instruction fetch unit. Once the instruction fetch unit supplies the instruction, the data portion of the token applied to the ALU input.

Figure 15 shows the relative timing for operation of the PMU. For clarity, the illustration shows operation of only one PMU where all incoming tokens were matched.

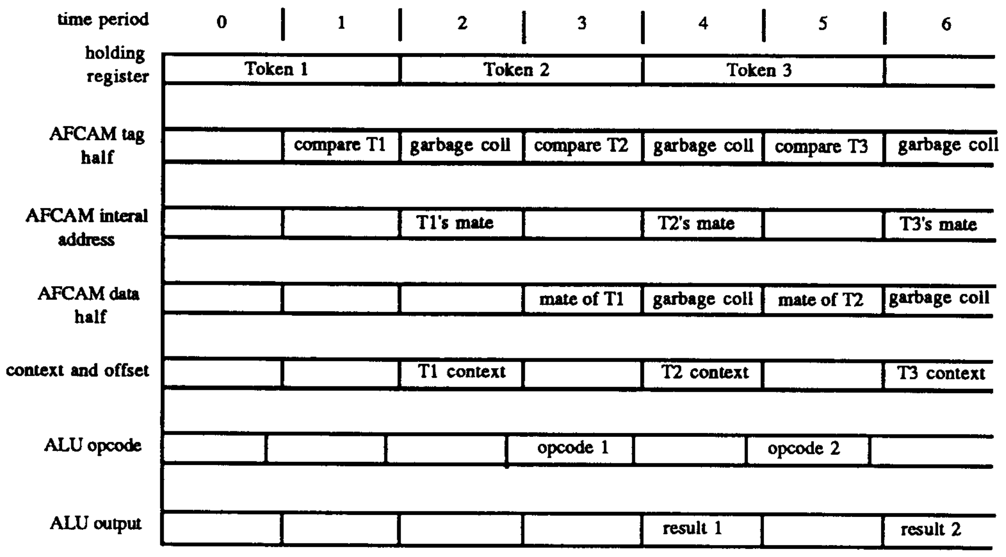


Illustration 15
Relative Timing for Operation of the PMU

A further level of detail is revealed here. Each AFCAM consists of two parts that operate in a pipelined fashion. The *tag half* does a comparison operation followed by a conditional store or garbage collect of the tag only. If a match occurs, the location of the mate is latched at the end of the compare cycle. During the tag half's garbage collect cycle, the *data half* accesses the data part of the matched token using the latched address. The next cycle will start another comparison cycle for the tag half while the data half garbage collects.

Indication of matching occurs at the end of the comparison cycle. If a match occurs, the context and offset of the incoming token are transferred to the instruction fetch unit. During this time, the access to the stored matching token is occurring. Thus during the next cycle, both operands and opcode arrive at the ALU. In the next cycle the ALU delivers the result.

With only one token cache operating, the ALU can emit a result every other cycle. With a second token cache would operating during the idle cycles of the first, the ALU could be kept supplied with operands and opcodes every cycle.

3.3 AFCAM Logical Operation

A storage map representation of the AFCAM can be used to graphically describe its operation. Below in illustration 16 we see a partially filled AFCAM with a write pointer pointing to the next available (empty) location. Note that tag and data are stored together in the same word line. Also shown are the input data path and overflow data path.

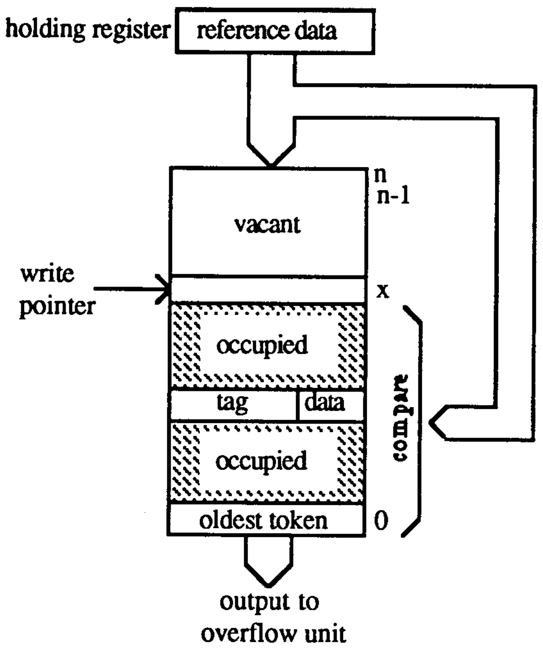
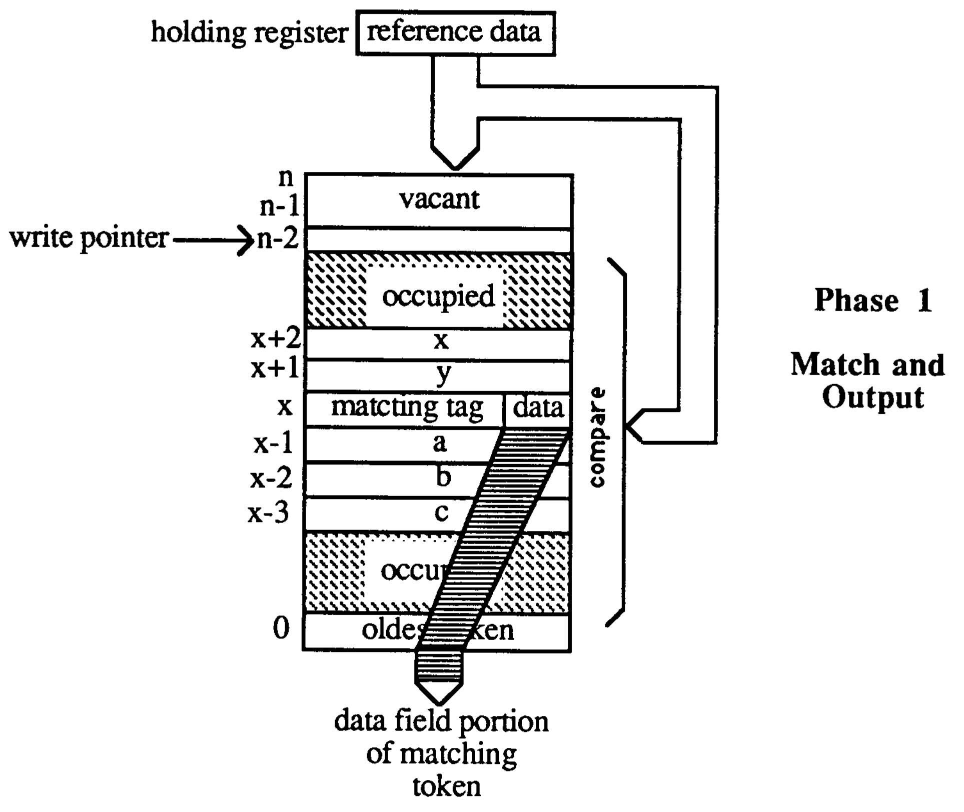


Illustration 16
AFCAM Logical Storage Map

A token entering the AFCAM is temporarily stored in the holding register. Then its tag is simultaneously compared against all other tags stored in the AFCAM. If the token's tag does not match any other tag in the AFCAM, the token is written into location x and the write pointer is incremented.

If the token's tag does match another tag in the AFCAM, a two-phase operation results. Illustration 17 depicts this situation. In phase 1, the reference data tag matches a tag in the AFCAM causing the data portion of the matching token to be output. The stored tag is not needed since a copy of it is stored in the holding register. In phase 2, the location that held the matching token is garbage collected. The garbage collection maintains FIFO ordering even though the matching token is in the middle of the storage map. The FIFO is thus articulated. The locations "above" the matched location are shifted down one location while the locations below the matched location remain stationary. The write pointer is then decremented by one location.

Should the AFCAM be full, and the input reference data fails to match any entry, the token in location 0 (the oldest token) is output to the secondary matching unit. Then the entire contents of the AFCAM are shifted downward freeing location n for the input data. Then the write pointer is decremented and the new token written in location n.



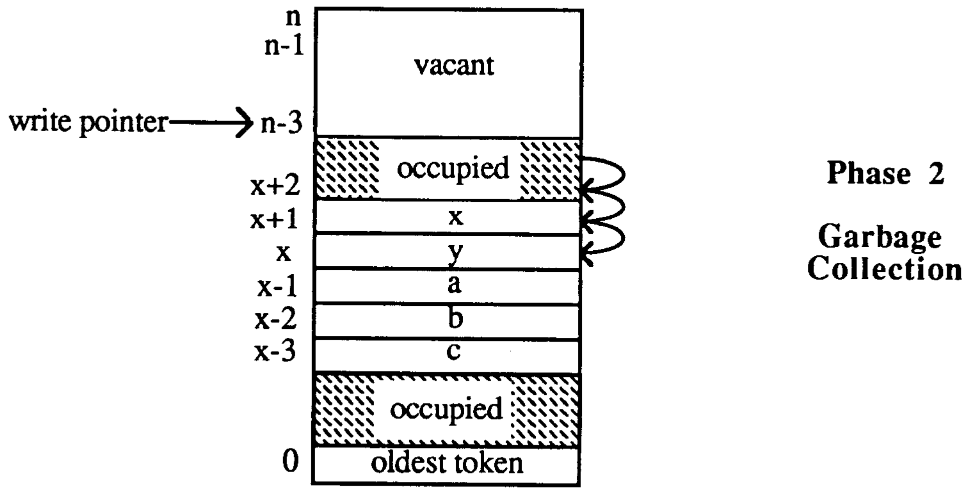


Illustration 17
Matching and Garbage Collection in the AFCAM

The AFCAM thus has two main attributes. First, it associatively matches tokens. Secondly, it maintains a time-ordered storage of tokens to implement a replacement policy where the oldest token is migrated to the secondary matching unit. The AFCAM is a hybrid of a CAM and a conventional FIFO. The observant reader probably foresees the difficulty in building such a device. This will now be discussed.

3.4 AFCAM Cell Structure and Operation

The AFCAM is the heart of the PMU. Vital to proving the feasibility of the hierarchical W-M unit, is a physically realizable AFCAM structure. In this section we will look at the structure of the AFCAM cell and its origin. Sufficient detail will be provided to prove the feasibility of the AFCAM cell design.

3.4.1 The CARM

In 1985, Matsushita Electric Industrial Co., Ltd., researchers described an 8-K bit (256 word by 32 bit) content-addressable and reentrant memory (CARM) specifically designed to be used in the waiting-matching unit of a dataflow computer [Kadota 85]. This device provides associative tag comparison with conditional garbage collection or data storage depending upon the occurrence of a tag match. The CARM's garbage collection algorithm does not maintain any ordering of the stored data. However, the basic CARM cell structure provided the starting point for development of the AFCAM cell.

The typical six-transistor static random access memory (SRAM) cell forms the basis for both the CARM and AFCAM storage cells. Four additional transistors are added to create an Exclusive-NOR for performing the bit-wise associative comparison. See illustration 18 below.

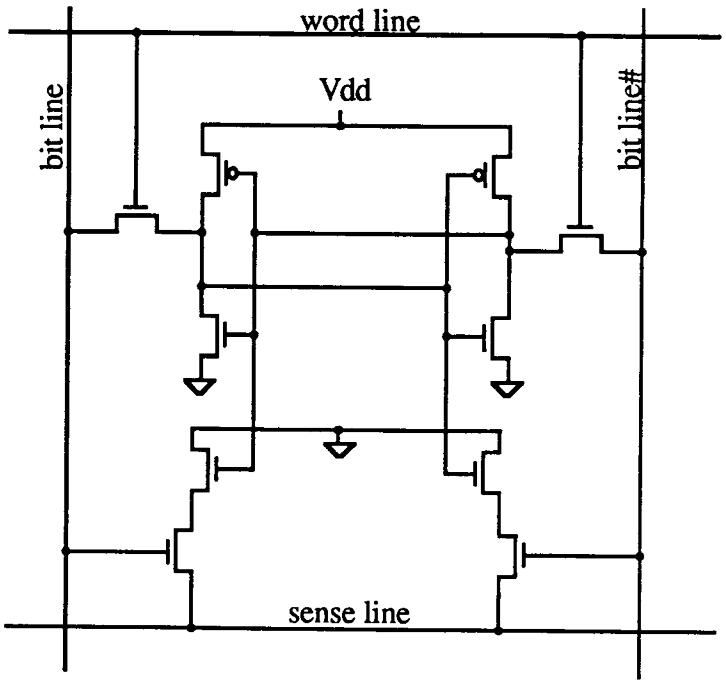


Illustration 18
CARM Storage Cell

During a typical read or write operation, the cell operates as an SRAM would. The appropriate word line is asserted and the bit lines are sensed or driven in a complementary fashion to read or write the cell. The "#" symbol indicates an complementary signal. During an associative comparison, the sense line is charged to the power supply potential and then the bit lines are driven from a data register as in a write operation. If a cells contents do not match the corresponding bit of the word in the data register, the Exclusive-NOR circuit creates a path to ground to discharge the sense line. If a match occurs for all cells (and thus bits) in a word, the sense line maintains its charge. This is sensed as a match condition and the address of the matching word is output to the outside world.

The CARM is fabricated in 2um CMOS technology with two-layer metalization. The die size is 4.8 x 7.1 mm2 and uses 99,000 transistors. At its minimum cycle time of 100ns the device dissipates one-half watt.

3.4.2 AFCAM Cell Structure

Below in illustration 19 is the proposed AFCAM cell. It resembles the CARM cell in that its basic structure is also the 6 transistor RAM cell with an Exclusive-NOR circuit. Added however, are three pass transistors (labeled a,b and c), and an extra inverter which allow a single-cycle word-adjacent shift. Also shown are two control lines, transfer# and transfer_pulse for activating this function. The pass transistors and control lines provide the mechanism that does the garbage collection while maintaining FIFO ordering.

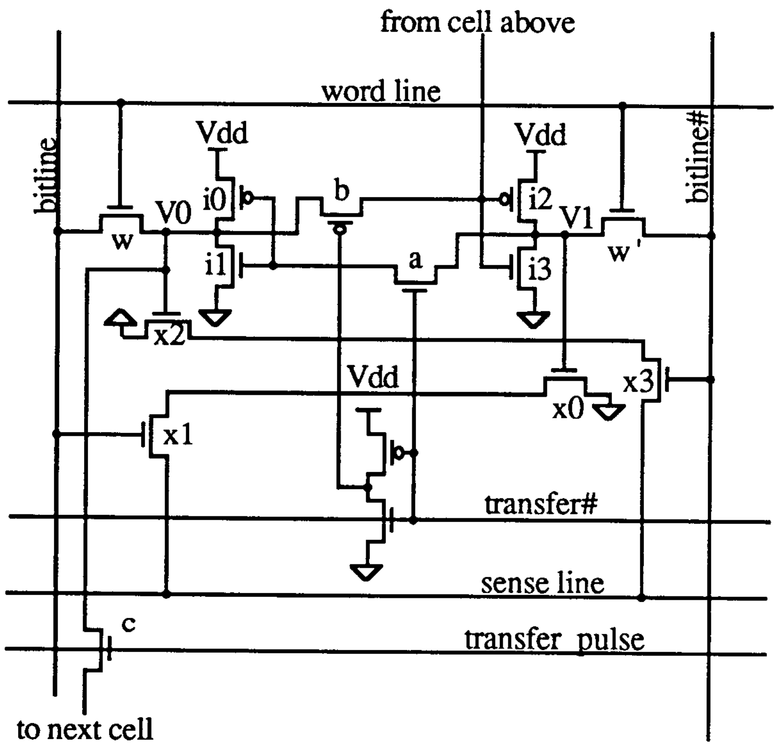


Illustration 19
Proposed AFCAM Cell

3.4.3 AFCAM Cell Operation

3.4.3.1 Associative Matching

When an incoming token arrives at the holding register, it is compared against all entries in the AFCAM. To do this, the word line, transfer# and transfer_pulse lines are held inactive. Transistors a and b are 'on' and transistor c is 'off'. See figure 24. The sense line is then charged (also known as *precharging*.) to the supply voltage level. Then, each bit of the token in the holding register drives the corresponding bit lines of each word in the AFCAM in a complementary fashion.

Transistor pairs *i0,i1* and *i2,i3* form two inverters which when cross coupled via transistors a and b form the storage cell. The complementary storage cell outputs are at points *v0* and *v1*. If the cell is holding a value of logic '1', then *v0* is Vdd and *v1* is at 0V. If the bit in the holding register corresponding to this cell is a logic '1', then when the bit lines are driven, *bitline* will be at Vdd and *bitline#* will be at 0V. If this is the case, *x0* is cut off by *v1* being at 0v and *x3* is cutoff by *bitline#* being at 0V. The discharge path for the sense line is thus broken indicating that for this bit position, the value in the cell equals the value in the holding register.

The same broken discharge path is created by complementary values on both the bit lines and in the storage cell. However, if the driven states of the bit lines and the cell contents differ, either the *x0,x1* or the *x2,x3* path will be established to discharge the sense line indicating that this bit and thus this word does not match the token in the holding register.

3.4.3.2 Writing into the Cell

If a matching cycle indicates that the word in the holding register does not match any stored word in the AFCAM, then the word must be written into the AFCAM. To do this, the transfer line and transfer pulse lines are held inactive. The sense line level is immaterial. Transistors a and b are 'on' and transistor c is 'off'. Then, each bit of the token in the holding register drives the corresponding bit lines of each word in the AFCAM in a complementary fashion as in the matching operation case. The word line is asserted, turning on transistors w and w'. This electrically connects the bitlines and the storage cell outputs and forces the value of the bitlines upon the storage cell outputs. The storage cell inverters are designed so that they may be overdriven by the bitline driver transistors should the holding register bit and cell values be different.

3.4.3.3 Garbage Collection

If the matching operation indicated a match between the token in the holding register and a word in the AFCAM, the location holding the matching token must be garbage collected and the next and succeeding younger tokens must be shifted downward so as to coalesce the storage map and maintain FIFO ordering. The ability of the AFCAM to do this operation in a single cycle is a unique feature.

Garbage collection must be done in one cycle to enable the AFCAM to process one token every other cycle. To do this, a simultaneous shift of all locations above the matching word must be done. To do this without an intermediate stage between words in a bit column, the cross coupled inverter pairs are temporarily split into two disconnected halves. One half holds the old value of the cell for transfer to the next cell down. The other half receives and holds the new cell value from the preceding cell. Three pass transistors and their control lines control the transfer.

To determine the boundary between words to be shifted and those to remain stationary, an address encoder and word row comparators could be used. The address encoder has as its inputs the sense line from each word row. When enabled, the address encoder outputs the binary coded address of the matching word row. The word row comparators are used to compare the matching word row address with their respective address. The output of the comparators indicates if that word row is greater than or equal to the matching word row.

See illustration 20 below showing the same bit of two adjacent words in the AFCAM.

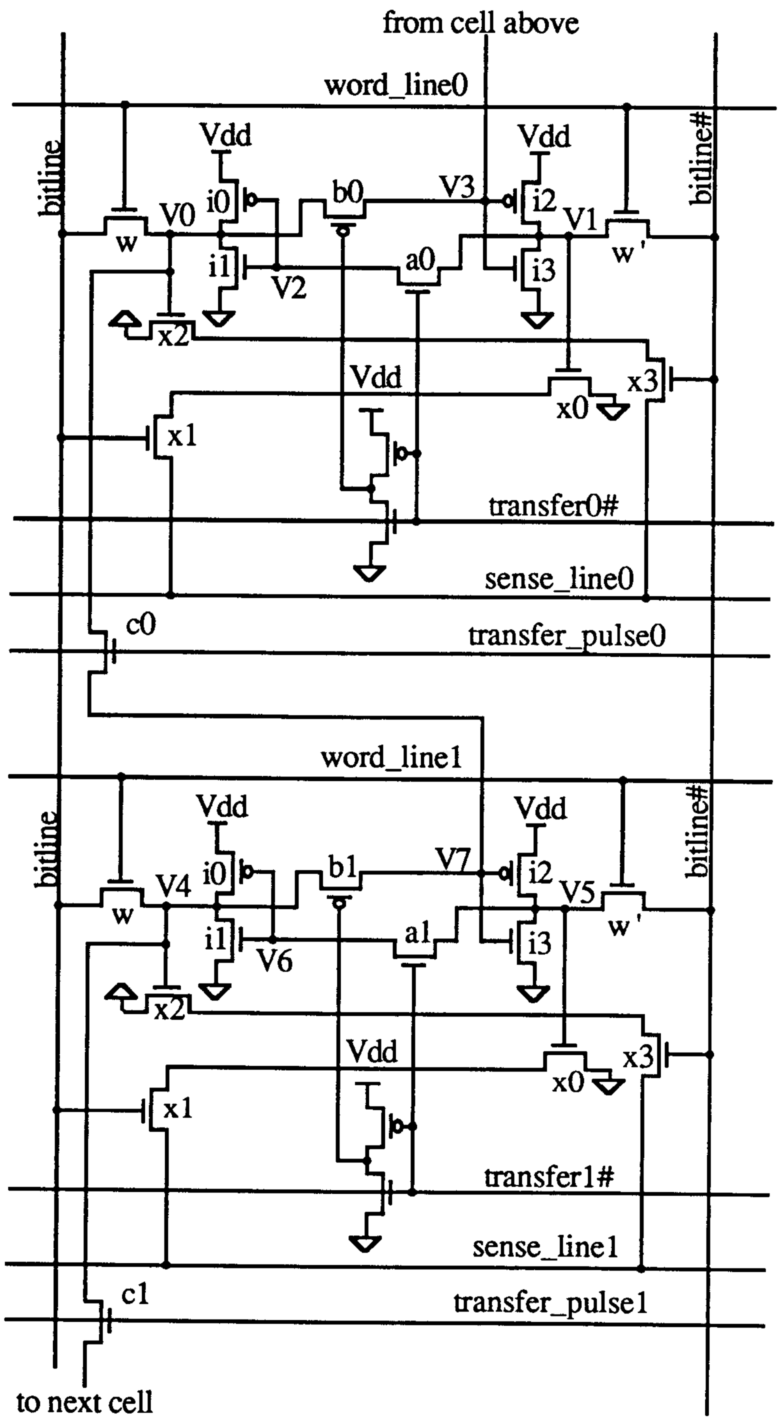


Illustration 20

Bit Slice of the AFCAM

Let's assume the case where the lower cell (word) matches the token stored in the holding register. *Sense_line1* will be in the logic '1' state indicating a match condition. This causes the address encoder to output the address of cell 1. This address is applied to the comparators associated with each word row. The output of the comparators for the lower word and the upper word indicate that they are located at an address greater than or equal to the matching word. The comparator output and the sense line drive the logic which generates the transfer and transfer_pulse outputs for each word row.

The first step after the match has occurred is to let all the comparator outputs settle out. After that a word row knows if it is involved in the shifting operation. For this example, the comparator for the upper word indicates the matching word is below or at this location. But its match line is low indicating it is not the matching word. It will thus receive data from the cell above it and shift its data down to the cell below it. The lower word's comparator also indicates the matching word is below or at this location.. However, because its sense line indicates a match, it is the matching row. The high state of the comparator will allow the row to be updated (garbage collected) but the high state of the sense line will inhibit the transfer of its data to the word row below it.

Data shifting begins by asserting *transfer0#* and *transfer1#* on those cells which will be receiving new data and transferring old data to the cell below. On the matching word row, only the transfer line is asserted. This action breaks the feedback path between the cross coupled inverters allowing them to hold different or identical values. While the feedback path is broken, the inputs to both inverters are left disconnected. The present state of the inverter is maintained by virtue of the gate capacitance at the input to each inverter pair.

The next step is to pulse *transfer_pulse0* for the upper word. This turns on transistor c0, connecting the output of the upper word's left side inverter to the input of the lower word's right side inverter. Since the right side inverters state is maintained only by its tiny gate capacitance, the output of the upper word's inverter easily charges or discharges it. At this point the right side inverter of lower word has assumed the old value of the upper word's right side inverter. Now the transfer pulse is removed.

The transfer line is now deasserted to reconnect the cross coupled inverter pairs. However the order in which the inverter pairs are reconnected is vitally important. For example, take the lower cell's right hand inverter. It how holds the old value of the upper word's right hand inverter. If however transistor b1 turns on before transistor a1, the left hand side inverter will drive the right side back into the old state of the cell. Transistor a1 must close first. This will force the left hand side inverter to assume the correct state. The correct ordering for the closing of the feedback path is assured by proper selection of pass transistors a1 and b1 and the inverter driving a1.

3.5 AFCAM Cell Implementation and Simulation

The proposed AFCAM cell was designed on paper until logically it was thought to be correct. The circuit was then simulated at the transistor level using a Intel proprietary analog simulator called iSPEC. All modes of operation were simulated under voltage, temperature and variations to prove the design. The AFCAM cell was designed with an advanced CMOS technology with minimum gate lengths of 0.9um.

Considerable experimentation was required to correctly determine the relative sizes of the transistors. Most critical were the sizes of the pass transistors and the inverter pair transistors. Pass transistors exhibit a phenomenon known as clock feedthought. This is where the signal on the gate of the transistor is feed through to the output of the transistor due to the charge transfer across the gate capacitance. This effect is more pronounced as the transistor size increases. Unfortunately, to effect the most efficient transfer across the transistor, it must be large in area and thus low in resistance.

The specific problem encountered with the AFCAM cell was that when the feedback path across the inverter pairs was broken, the clock feedthrough was of sufficient magnitude to almost discharge the very small gate capacitance of the inverter transistors. If the gate capacitance of the inverter transistors was made larger by increasing their size and thus decreasing their susceptibility to clock feedthrough, the pass transistor's series resistance limited the rate at which the inverter gate capacitance could be charged, greatly limiting the speed of the shift down operation. This required careful matching of the sizes of the pass transistors and the inverter transistors to allow a maximum drop of gate voltage of 20% of Vdd.

The selection of the pass transistor type was also tricky. See Figure 20. Initially the cross coupled inverter pass transistors were chosen to be P-channel devices and the transfer lines were asserted high. The P-channel transistors exhibited the least clock feed through. The size of the transistors was adjusted so that a0 was big (fast) and b0 was small (slow). But the simulation indicated that node V1 never acquired the new shifted down value but rather the old value of V0. Increasing a0 to much larger sizes did not speed it up enough and the clock feedthrough was increasing proportional to the transistor size.

The next try was to use keep a0 a P-type device and use a small N-channel transistor with an inverter in series with its gate for device b0. The inverter would give the correct sense of the transfer0 line to the transistor while further increasing the delay. This also did not work much to my amazement. Node V1 acquired the old V0 value.

Further examination of what was happening cleared the confusion. The path from V1 to V3 must be faster than the path from V0 to V3. In other words, the path though two inverters must be faster than the path through one transistor.

The last simulation showed that the small N-channel transistor driven through an inverter was faster than the bigger P-channel transistor that was being driven directly from the transfer0 line. This observation led to the solution. Just reverse the position of the two transistors and the inverter. The final solution for the fast leg of the feedback path was to use the small but fast N-channel transistor and drive it from an inverter connected to transfer0.(remember at this time transfer0 was still active high asserted) This scheme has the advantage of lessened clock feedthrough due to a small pass transistor and requiring less silicon area. The slow leg was made up from a small P-channel transistor.

In CMOS technology, PMOS transistors are roughly one-third as fast for a given size as a NMOS transistor. Thus the transistor that must close first (a0) was chosen to be NMOS. The slow transistor (b0) was chosen to be PMOS. Curiously enough the turn-on of a0 is much faster even with the inclusion of the inverter in series with the drive from the sense line. The final selection of drive transistors and inverter minimized clock feedthrough and silicon area used.

Two further improvements were made to the transfer transistor configuration. To insure a greater margin of safety in transferring data from one cell to another, the polarity of the transfer lines was inverted, and the inverter was put in series with the gate of the P-channel transistor. This adds delay in the path that is to be the slowest and removes the inverter delay from the fast path. Furthermore, the transistor sizes in the inverter driving the P-channel transistor were set to have a threshold of about 65% of Vdd. By using the slew rate of the transfer signal, the high threshold of this inverter further delays the turn-on of the P-channel transistor. This latest version of the cell was what was used in the final simulations.

At this point the cell was working well. However, observation of the voltages at the inputs to the cross-coupled inverters showed that the inverters were just able to make correct transitions. This happens in two cases. Whenever the N-channel transistor tried to transfer a Vdd level to the input of the inverter pairs, it could only reach about 3.5 volts. Likewise, when the P-channel transistor tried to transfer a 0V level to the input of the inverter pairs, it could only pull down to about 0.9 volts. This effect is due to the close proximity of the source to body voltages in each of the transistors.

The threshold of the inverters was at that time about 1 volt. To insure a greater safety margin, the threshold level of the cross coupled inverter pairs was increased to a point mid-way between 3.5 and 0.9 volts. The actual threshold is about 2.5 volts.

Three scenarios were simulated to verify the operation of the cell. These scenarios were:

1. Cell write.
Each of the cells are written with an opposite value as they initially had.

2. Match operation.
The sense line is precharged while the bit lines are set to zero volts. Then the bit lines are changed to cause a mismatch to the top cell, and then the bottom cell.

3. Garbage collection (shift down of data).
Top and bottom cells receive opposite values they initially had. Top cell initially holds logic one, bottom cell logic zero. Top cell is written into by previous cell and bottom cell gets data from top cell. Then the top cell is written again by the same value from the previous cell with no resultant change. By initalizing cells to opposite states, shift down of data of both polarities is observed.

All three scenarios were tried at:

nominal	Temp = 25 °C	Vdd = 5 volts
best case	Temp = 0 °C	Vdd = 6 volts
worst case	Temp = 85 °C	Vdd = 4 volts

Best and worst case simulations showed the cell still functioned. The simulation results under nominal conditions for each scenario may be found in Appendix A.

3.6 AFCAM Topology

As previously mentioned, there are two halves to the AFCAM; the tag half and the data half. The AFCAM cell just discussed resides on the tag side only. The other side holds the data that is associated with a given tag. Thus the tag and the data attached to the tag are located at the same word address or row in the AFCAM. See illustration 21.

At the end of the match operation, the value of each sense line is held by a latch located at the boundary between the tag and data sides of the AFCAM. The outputs of the latches are connected to the word lines of the corresponding word on the data half of the AFCAM. Once the value of a matching sense line is latched, the access of the data portion of the token may begin. This happens in parallel with the garbage collection operation. Most other CAMs and the CARM first encode the address of the matching word and output that value. Another chip takes the encoded address, decodes it, and accesses the stored data word. The AFCAM however, integrates both tag and data halves so that the sense line may directly drive the word line to access the data. This approach is much faster.

Since the tags in the AFCAM are never read out, only compared, the tag half of the AFCAM does not any need sense amplifiers on the bit lines to read the value of a cell. Sense amplifiers are a major consumer of power in a typical SRAM. Eliminating these amplifiers will ease the power consumption significantly.

Since the cell inverter pairs never have to drive the highly capacitive bit lines, the inverter pair transistors may be made smaller than in a conventional SRAM cell. The inverter pairs only need to drive the inputs to the exclusive NOR which consists of a single MOS gate. The smaller inverter pair transistors also make it easier for the drivers on the bit lines to force them into the proper state when the cell is written, further reducing power.

The data half is never involved with the matching operation, thus it does not need the Exclusive NOR circuitry. But the data must be shifted with the tags, so the shift down circuitry must be included. In addition, sense amplifiers will be needed since the data half requires both read and write access.

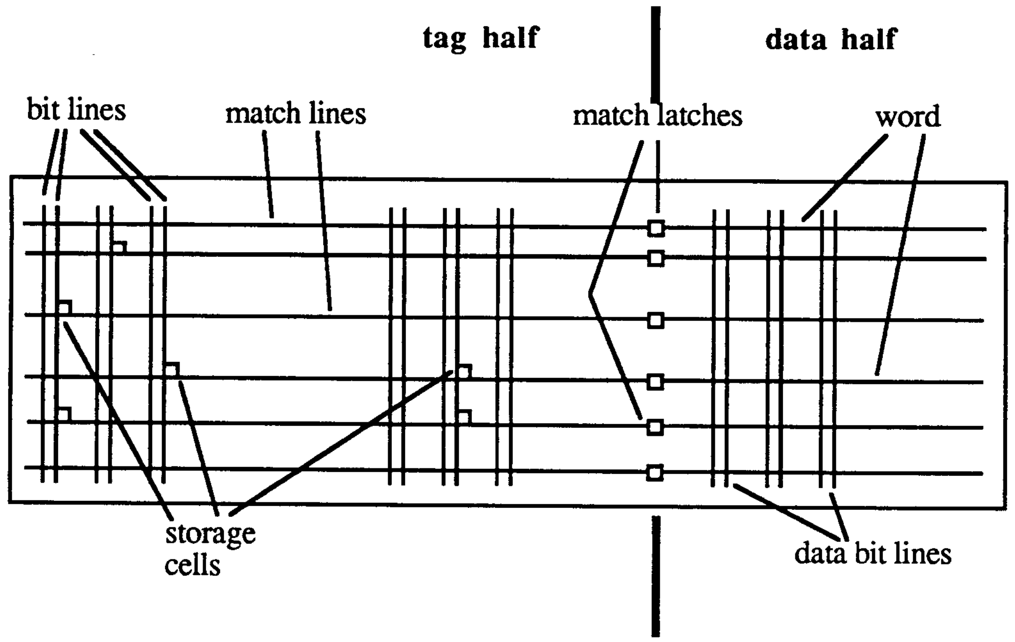


Illustration 21

AFCAM Physical Organization

3.7 Feasibility of Integration

Considering the complexity of the AFCAM cell it is necessary to determine the feasibility of putting the PMU and an ALU on a single silicon die. Hopefully, there would be additional room left over to implement the other functions of a PE. The PMU and ALU must be on the same die to keep this high bandwidth path running at full on-chip speeds. The secondary matching unit due to its size must be external to the die. This is no problem because of the relaxed access time needed to the SMU.

The original CARM cell consisted of 10 transistors. The AFCAM tag cell is constructed of 14 transistors while its data cells are made of 10 transistors. (Data cells do not need the exclusive NOR) For two AFCAMs with a total of 256 token storage locations, each holding a 48 bit tag and a 32 bit data word, the storage cells would require 172,032 transistors. Not included are the sense amplifiers, address encoders and decoders, magnitude comparators and other logic. Assuming these take up another 20,000 transistors, the total transistor budget would be less than 200,000 transistors.

Intel's 80860, an advanced numerically optimized RISC microprocessor, required 250,000 transistors to implement its scalar and floating point ALUs. If these units were merged on a chip with a PMU, the total transistor budget would be approximately 450,000 transistors.

As an example of the level of integration available in 1991, the Intel 80860XP microprocessor built with 0.9um technology has 2.5 million transistors. Even from a gross approximation using only transistor count, the PMU, and an ALU should fit comfortably on a reasonably sized die. It is certainly conceivable that the Instruction Fetch, Compute Tag, and Form Token units would also fit on the die.

3.8 Secondary Matching Unit

In a memory hierarchy, the greater the hit rate of the primary storage, the less the secondary storage will contribute to the average access time. Thus for high primary storage hit rates, the speed of the secondary storage is not as important. This situation gives the designer greater flexibility in the design of the secondary memory.

We will choose as a working example the matching unit of the Manchester dataflow machine. Although it was built with late 1970s technology, its inclusion in a hierarchical W-M unit of the type just described would give reasonable performance. If implemented using today's technology, the performance of this W-M unit would easily double.

The Manchester W-M unit was implemented with Metal Oxide Semiconductor (MOS) Dynamic Random Access Memory (DRAM) memory devices with an access time of 150 ns. Using these memories, the unit could execute an successful search in 240 ns and an unsuccessful search in 320 ns. These figures compare favorably with the 240 ns cycle time of the devices used [da Silva 83].

Due to advances in memory technology, the density of available Static Random Access Memories (SRAMs) is greater than the DRAMs used in the Manchester machine. Thus a W-M unit like the Manchester unit could built using these memories instead. The cycle time of current SRAMs is equal to their access time which is about 20ns. Using these memories, a conservative estimate of the time to do a successful and unsuccessful search in a Manchester style W-M unit would be 60ns and 80ns respectively.

4. System Performance with the Hierarchical W-M Unit

4.1 Hit Rate

The performance data for the hierarchical W-M Unit described is derived from the research done at MIT by Steven Brobst [Brobst 87]. He modeled the performance of the TTDA using a token cache of various sizes and varied loop bounding sizes. His token cache was modeled as a fully associative CAM. Transfers out of the cache (my PMU) to secondary storage (SMU) was one token with a strictly FIFO replacement policy. Mr. Brobst made no assumptions of memory unit speeds but instead only measured the percentage of tokens which found mates in the cache versus the secondary memory.

The hit rate simulation was based on one iteration of the Simple Code on a 10x10 mesh. The cache size and loop unbounding were varied to obtain a family of curves expressing the percentage of matches occurring in the token cache. The results of the simulation may be seen in illustration 22.

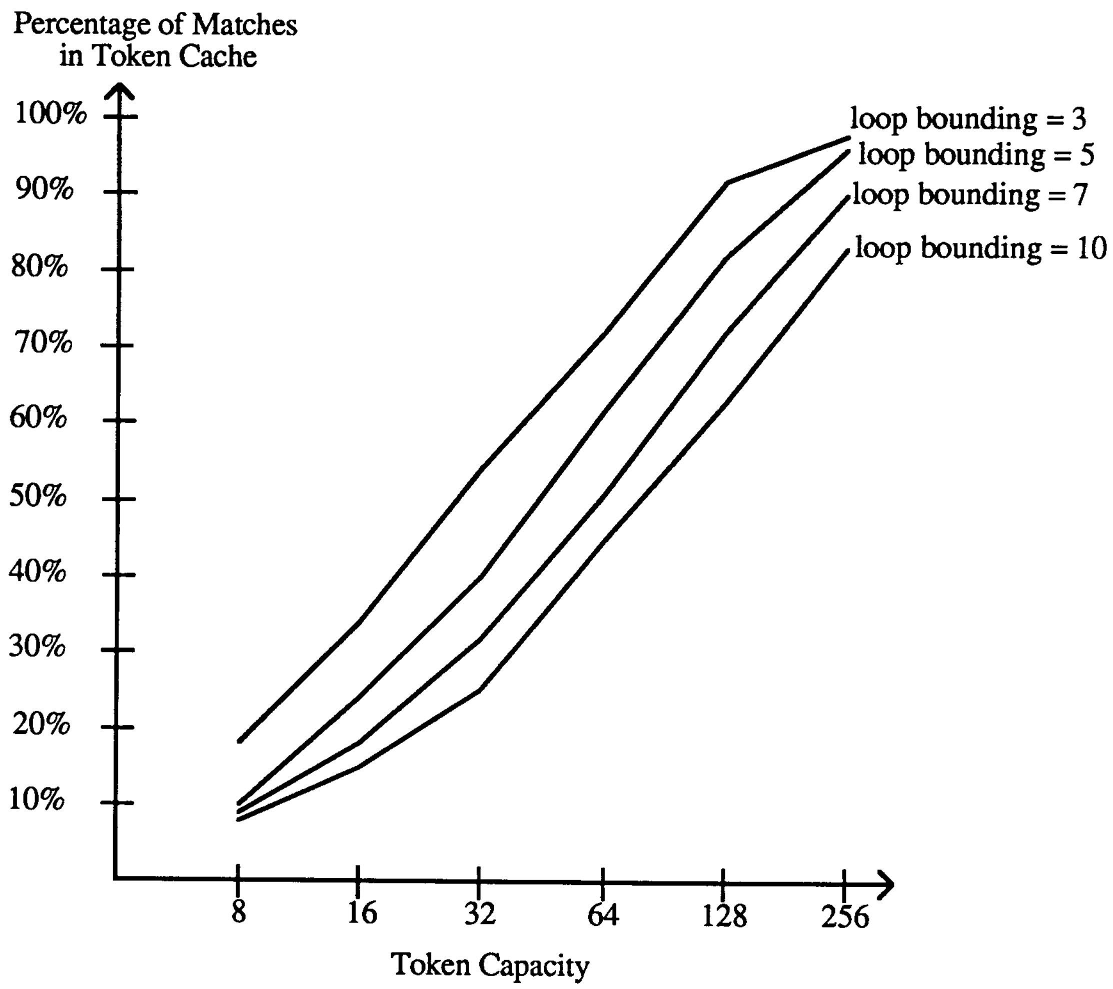


Illustration 22
Token Matching Hit Ratio for a FIFO Caching Policy [Brobst 87]

Illustration 22 shows that for even modest cache sizes, the token hit rate is quite good. An important observation is the strong effect of controlling the loop unbounding upon the hit rate. For a cache size of 256 tokens when loop unbounding was set to 10 (unbounded unfolding on the 10x10 mesh) the hit rate is adversely affected by 15%.

To measure the performance of the W-M unit, we compute its average access time using the hit rate data from the graph. A conservative estimate for the access time for a token in a match condition from the AFCAM is 25ns. This is based on the AFCAM cell simulations that indicate that:

Phase one operations (match and access data):

precharge/match could be done in about 8ns

stored data item whose tag matched can be accessed in about 10ns

Phase two operations (garbage collection) :

encoding and comparing address would take about 6ns (estimate not supported by simulations)

shift down data can be done in about 11ns

Typical SRAMs have a 20ns access time. The match to output time for the secondary matching unit has been estimated to be 60ns. With these figures and loop unfolding set to three, we get:

(hit rate * access time of primary storage) +
(hit rate * access time of secondary storage) = average match to token output time

0.98 x 25ns +
0.02 x 60ns = 25.7 ns average match to token output time

This is of course using two AFCAMS within the PMU. After accounting for the fact that fifty to sixty percent of tokens coming into the PE are monadic, the hierarchical W-M unit should easily be able to keep a ALU with a 20ns cycle time supplied with tokens. This should effectively remove the bottleneck at the W-M Unit.

V. Conclusion

Dataflow computers hold great promise for delivering extremely high levels of performance required in the future. They have already exhibited linear speed-up with increased numbers of processing elements for some problems. Dataflow computers were designed to support languages that naturally express parallelism. As such, they can efficiently find all the parallelism within a program, thus yielding the maximum attainable speedup.

A weak point in dataflow computers built so far is the performance of the W-M store. Research has shown that the W-M unit is the bottleneck in the hardware design. This is no surprise, as the bandwidth and storage requirements of the W-M unit are formidable. W-M units with pseudo-associative matching using hashing algorithms have been built, but their performance is far from satisfactory.

There have been a number of recent advancements in the design of content addressable memories. One Japanese company has built an 8K-bit content addressable memory that could be used in a dataflow computer. Its small size would limit it to use in only toy programs however. At this point in time, it does not seem that a reasonably sized, non-hierarchical W-M unit can be placed on a single chip. Such a chip would have to store millions of bytes and be able to perform sophisticated associative matching and garbage collection on each location.

Fortunately, tokens arriving at the W-M unit in a dataflow computer exhibit strong temporal locality. Thus, large numbers of tokens that have just arrived can expect their mate to arrive within a short time. This suggests the use of a technique used in conventional computers, a memory hierarchy. In the dataflow context, the subset of tokens whose mate will arrive soon are kept in a fast token cache where they may be matched quickly. The other tokens migrate out after "aging" momentarily in the cache.

Using an existing CAM design as a basis, a hierarchical W-M unit has been described that uses a fast, 256 location token cache to match tokens from active code blocks. The heart of the token cache is a new memory cell designed in an advanced CMOS technology. Electrical simulation results indicated that the AFCAM cell should work and furthermore should be able to match and deliver tokens to an on-chip ALU at rates sufficient to keep it fully utilized. A slower and much larger secondary store holds tokens overflowing form the token cache.

Simulation results of dataflow computers using such a W-M unit are very encouraging. The probability of a token finding its mate before migrating out of the token cache is better than 98% for a 256 location token cache. The hierarchical W-M unit can provide both the speed and capacity needed to create a much higher performance dataflow machine than has yet been built. The size of the token cache unit is also small enough to fit on a single chip with an integer and floating point unit.

Bibliography

[Ackerman 82]	William B. Ackerman Data Flow Languages *IEEE Computer*, February 1982, pg. 15 - 25
[Arvind 78]	Arvind, Kim P. Gostelow, Wil Plouffe The (Preliminary) Id Report: An Asynchronous Programming Language and Computing Machine Technical Report #114 Department of Information and Computer Science, University of California, Irvine, May 10, 1978
[Arvind 80]	Arvind, Robert E. Thomas I-Structures: An Efficient Data Type for Functional Languages University of California, Irvine, CA June 25, 1980
[Arvind 82]	Arvind, Kim Gostelow The U-Interpreter *IEEE Computer*, February 1982
[Arvind 83]	Arvind, and R.A. Iannucci A Critique of Multiprocessing von Neumann Style *The Proceedings of the 10th Annual International Conference on Computer Architecture*, 1983, pgs. 426-436
[Arvind 86a]	Arvind, David E. Culler Dataflow Architectures MIT/LCS/TM-294, February 12, 1986
[Arvind 86b]	Arvind, Robert A. Iannucci Two Fundamental issues in Multiprocessing Computation Structures Group Memo 226-5, July 25, 1986 Laboratory for Computer Science, MIT
[Arvind 87a]	Arvind, R. A. Iannucci Two Fundamental Issues in Multiprocessing Computation Structures Group Memo 226-66, May 5, 1987 Laboratory for Computer Science, Massachusetts Institute of Technology

[Arvind 87b]	Arvind, R. S. Nikhil Executing a Program on the MIT Tagged-Token Dataflow Architecture Computation Structures Group Memo 271, March 1987 Laboratory for Computer Science, Massachusetts Institute of Technology
[Brobst 87]	Stephen A. Brobst Organization of an Instruction Scheduling and Token Storage Unit in a Tagged Token Dataflow Machine *Proceedings of the 1987 Conference on Parallel Processing*
[Bursky 91]	ISSCC Review: Digital Technology *Electronic DesignI* February 14, 1991, p55
[Cole 91]	Bernard C. Cole Japan's Vision of the Future *Electronic Engineering Times* February 18, 1991, pg 84
[Culler 87]	David Culler, Arvind Resource Requirements of Dataflow Programs Computation Structures Group Memo 280 December 15, 1987 Laboratory for Computer Science, Massachusetts Institute of Technology
[da Silva 83]	J.G.D. da Silva, I. Watson Pseudo-associative store with hardware hashing *IEE proceedings*, Vol. 130, Part E, No. 1, January 1983
[Dennis 80]	Dennis, J. B. Data Flow Supercomputers MIT Laboratory for Computer Science *IEEE Computer*, vol 13, November 1980, pg 48 - 56
[Ekanadham 87]	Kattamuri Ekanadham, Arvind SIMPLE: Part I, An Exercise in future Scientific Programming Computation Structures Group Memo 273,July 1987 Laboratory for Computer Science, Massachusetts Institute of Technology

[Gurd 85] J.R. Gurd, C.C. Kirkham, and Ian Watson
The Manchester Prototype Dataflow Computer
Communications of the ACM, January 1985, Vol. 28, No.1

[Hiraki 84] Kei Hiraki, Toshio Shimada, and Kenji Nishida
A Hardware Design of the Sigma-1, A Data Flow Computer for Scientific Computations
Proceedings of the 1984 International Conference on Parallel Processing, pg 524

[Hwang 84] Kai Hwang and Faye Briggs
Data Flow Computers and VLSI Computations
Computer Architecture and Parallel Processing Chapter 10
McGraw-HIll Book Co, 1984

[Kadota 85] Hiroshi Kadota, Jiro Miyake, Yoshito Hishimichi, Hitoshi Kudoh, and Keiichi Kagawa
An 8-kbit Content-Addressable and Reentrant Memory
IEEE Journal of Solid-State Circuits, Vol. SC-20, No. 5, Oct. 1985

[Kahaner 90] David Kahaner
ETL Dataflow Project, Revised and Updated
June 25, 1990
Posted communication from USENET
Article 1434 from comp.parallel

[Lerner 84] E. J. Lerner
Data-flow architecture
IEEE Spectrum, vol. 21, April 1984, pg. 57 - 62

[Lubeck 90] Olaf Lubeck
Los Alamos National Laboratory
ETL Dataflow Project, Revised and Updated
June 25, 1990
Posted communication from USENET
Article 1434 from comp.parallel

[Mead 80] Carver Mead, Lynn Conway
Introduction to VLSI Systems
Addison -Wesley, 1980

[Nikhil 86]	Rishiyur S. Nikhil, Keshav Pingali, Arvind Id Nouveau Computation Structures Group Memo 265, July 23, 1986 Laboratory for Computer Science, MIT
[Papadopoulos 88]	Gregory Papadopoulos Implementation of a General Purpose Dataflow Multiprocessor Doctoral Dissertation, Massachusetts Institute of Technology August, 1988
[Pierce 90]	Paul Pierce Intel Corporation, Supercomputer System Division Personal Communications, 1989
[Seitz 84]	Charles L. Seitz, Juri Matisoo Engineering limits on computer performance *Physics Today*, May 1984
[Traub 86]	Kenneth R. Traub A Compiler for the MIT Tagged-Token Dataflow Architecture August 1986 (Masters Thesis, MIT)
[Treleaven 82]	Treleaven, P.C., D.R. Brownbridge, and R.P. Hopkins Data-Driven and Demand-Driven Computer Architecture *Computing Surveys*, Vol. 14, No. 1, March 1982, pg. 93 - 143
[Treleaven 84]	Treleaven, P.C. Decentralized computer architecture *New Computer Architectures*, Edited by J. Tiberghien, Academic Press 1984
[Vegdahl 84]	Steven R. Vegdahl A Survey of Proposed Architectures for the Execution of Functional Languages Computer Research Laboratory, Tektronix, Inc. *IEEE Transactions on Computers*, Vol. C-33, No.12, December 1984

Appendix

Illustration 23 shows the schematic of the two AFCAM cells that were used in the simulations. This schematic is identical to illustration 20 with the exception of a few external devices that are used to properly stimulate the circuit.

The two P-channel transistors whose gates are connected to PCHG are used to precharge the sense lines SENSE0 and SENSE1. The N-channel transistor above the top cell is connected to a signal called PREV. PREV is just the value of the cell that would be above the top cell. The inverter at the bottom whose input is at node 11, simulates the load of the next cell below.

The symbol "#" indicates an asserted low (zero volt) signal. The numbers adjacent to each transistor indicates its channel width in microns. The channel length in this technology is 0.9 micron. The symbols N1, N2, N10 refer to circuit node numbers. The waveform printouts use these designations to identify individual waveforms.

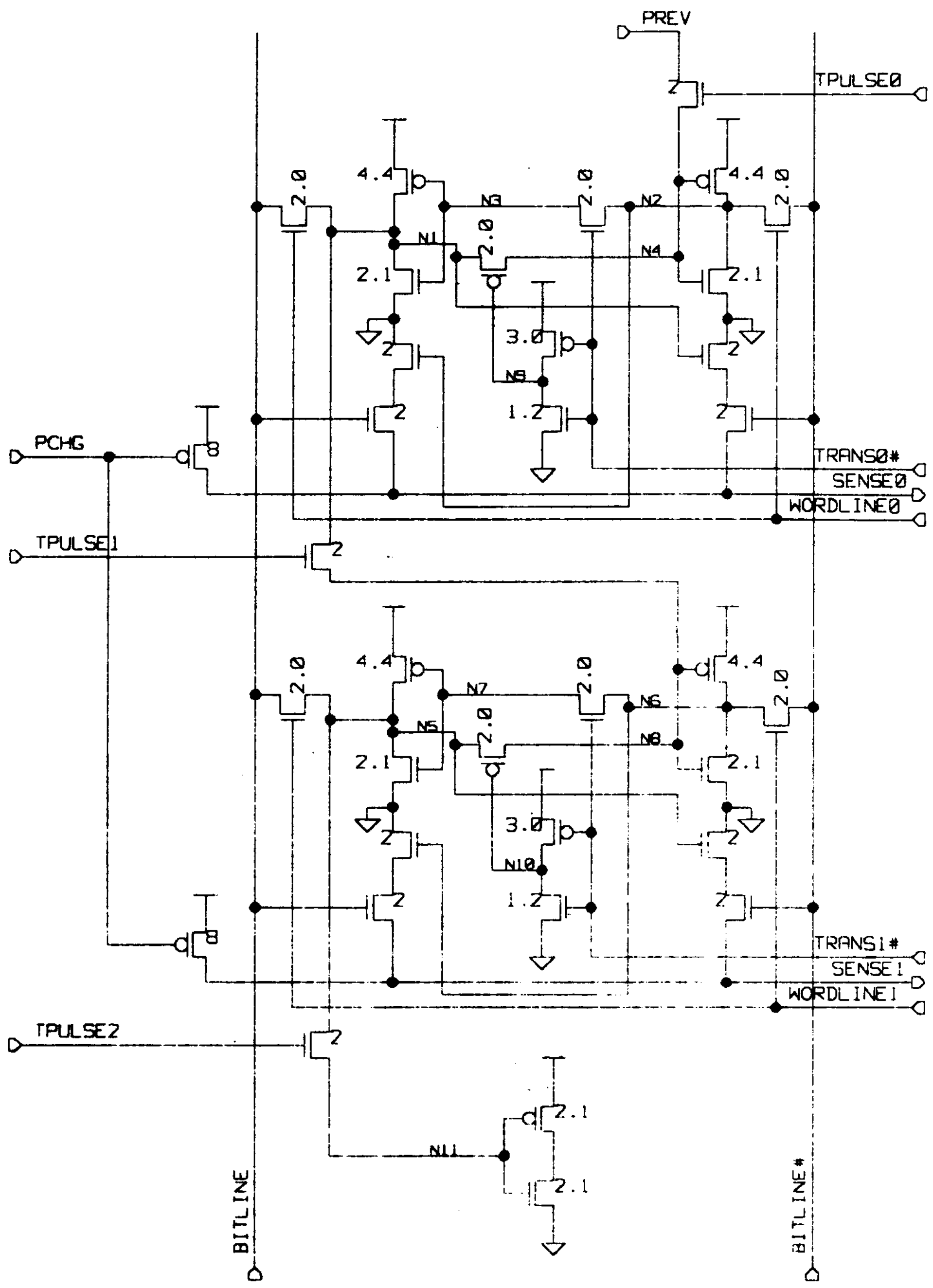


Illustration 23
AFCAM Cells Used in Simulation

Illustration 24 shows the results of the write cell test. This simulation was done with Vdd set to 5 volts, die temperature at 25 °C, and typical process. Node 1 (top cell) was initialized at Vdd, and Node 5 (bottom cell) at zero volts. The bitline was initialized at zero volts also. At 12ns, the top cell was written with the value on the bitline which caused node 1 to fall to approximately zero volts. Bitline was switched to Vdd at 20ns, and the bottom cell was written with the value on the bitline at 22ns. Accordingly, node 5 transitioned from approximately zero volts to Vdd. Both write operations completed within 3ns. The write cell test portion of the simulation concludes at 30ns.

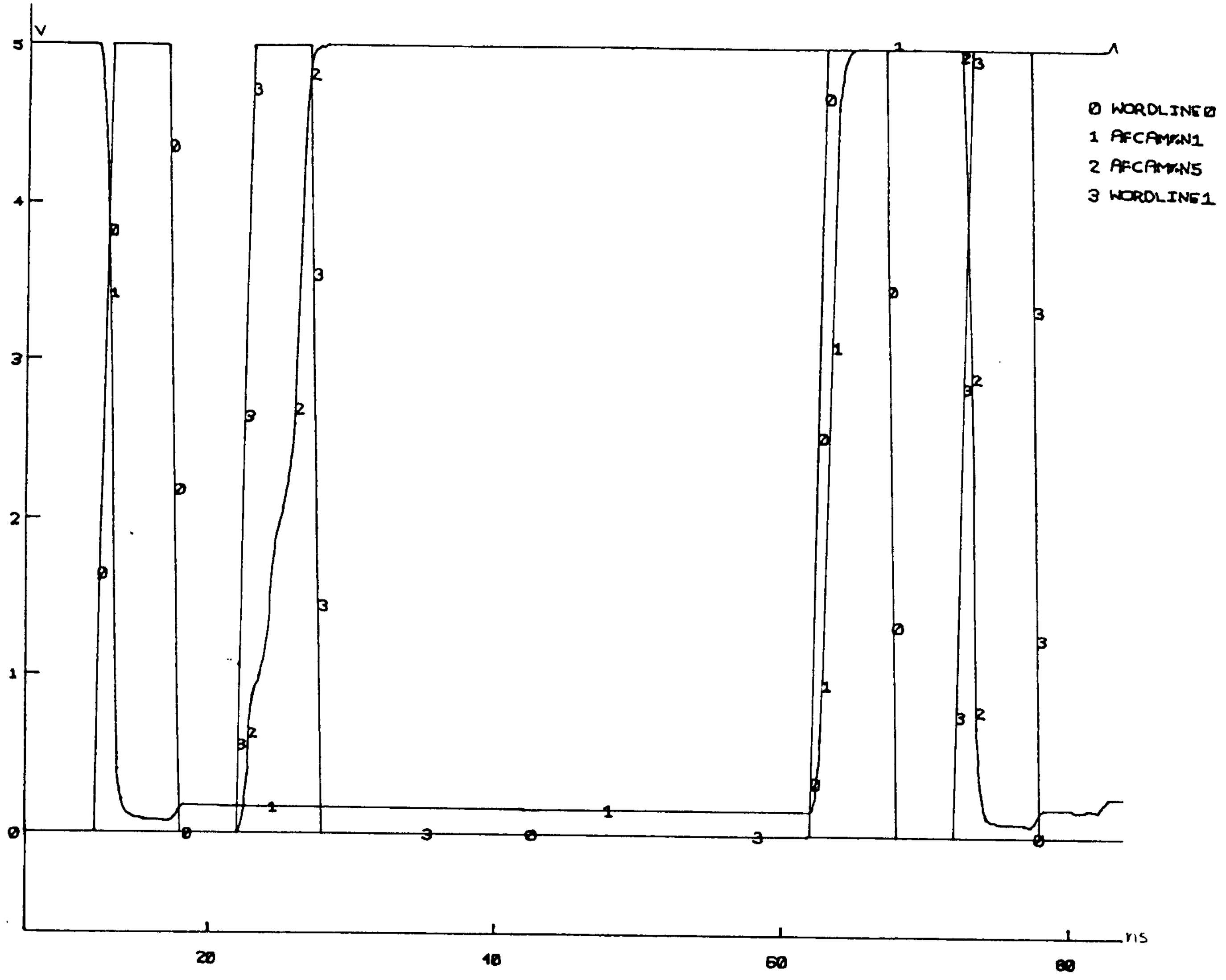


Illustration 24
Write Cell Simulation

Illustration 25 shows the results of the match test. This simulation was done with Vdd set to 5 volts, die temperature at 25 °C, and typical process. Following the write cell test, node 1 (top cell) is initialized at zero volts, and node 5 (bottom cell) at Vdd. Both bitlines are set to zero volts. At 32ns, the precharge signal PCHG is asserted low for 2ns. This results in SENSE0 and SENSE1 transitioning to Vdd. At 42ns, bitline transitions to Vdd while bitline# remains at zero volts. SENSE0 falls within 2ns to zero volts indicating a mismatch. The mismatch case is the important situation to look at here and not the match case because a match results in no change in the sense lines.

At 52ns bitline and bitline# swap states and the bottom cell mismatches resulting in SENSE1 transitioning to zero volts very quickly. The match test concludes at 55ns.

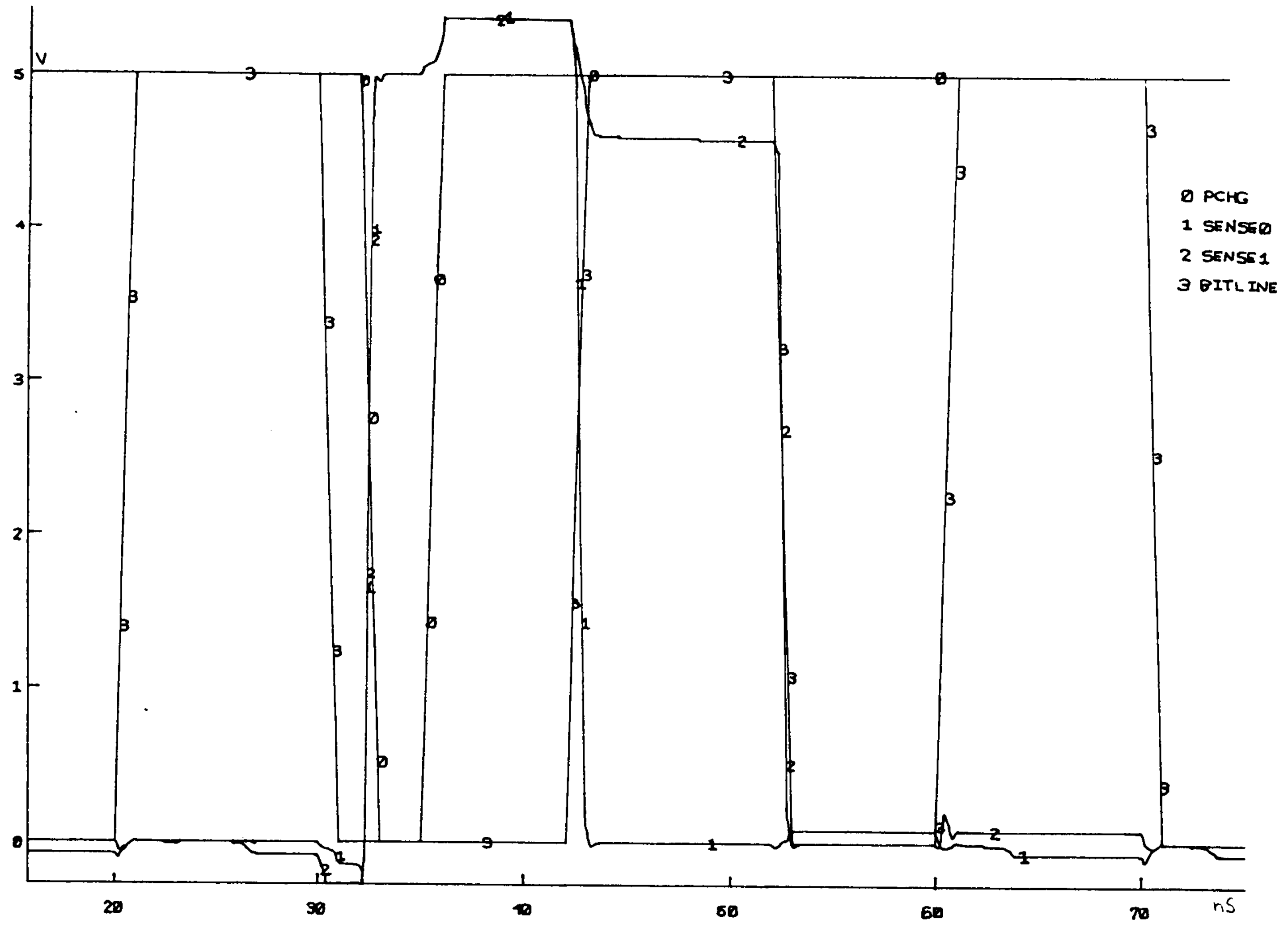


Illustration 25
Cell Match Simulation

Illustration 26 shows the results of the shiftdown test for the top cell only. This simulation was done with Vdd set to 5 volts, die temperature at 25 °C, and typical process. Following the match test, node 1 (top cell) is initialized at 0 volts, and node 5 (bottom cell) at Vdd. Top and bottom cells are first reinitialized by writing both cells to their opposite states. At the beginning of the shiftdown test node 1 is again at Vdd and node 5 is at zero volts. The signal PREV is fixed at zero volts.

At 82ns, TRANS0# is asserted low. This disconnects the cross coupled inverters in the cell. Note that nodes 3 and 4 are perturbed very little by the clock feedthrough at this point. At 84ns, TPULSE0 is asserted high to transfer the value of PREV (zero volts) into the right hand side inverter of the upper cell. This is seen as node 4 transitions from Vdd to zero volts at 85ns. The right hand side inverter's output correspondingly goes to Vdd.

During this time the output of the left hand side inverter of the top cell was connected to the input of the right hand side inverter of the bottom cell. The slight dip in the output voltage of the inverter (at node 1) is indicative of this action.

At 90 ns, TPULSE0 goes back to zero volts. Then at 92ns, TRANS0# goes back high, reconnecting the cross coupled inverters. The output of the right side inverter drives node 3 to 3.5 volts. This causes the left side inverter to switch state as node 1 assumes a new value of zero volts.

Note that in the same way that node 3 did not go to Vdd, node 4 did not go all the way to zero volts. This is because the gate to source threshold was shifting as the transistors outputs began to swing to either the Vdd or ground rails. This incomplete transition could cause problems if the cross coupled inverter pairs did not have their thresholds set midway between the levels of nodes 3 and 4.

A second half of the shiftdown test is conducted beginning at 102ns. This time nothing happens when an identical value is transferred from the value of PREV to the top cell. No transitions of the cell are noted, indicating correct operation.

Illustration 27 shows the operation of the bottom cell during the same time period as was just discussed for the top cell. The only difference is that the bottom cell was initialized oppositely to the top. Thus, when the shiftdown occurred, cell transitions from zero to one and one to zero could both be seen. The bottom cell operated correctly in a similar manner as the top.

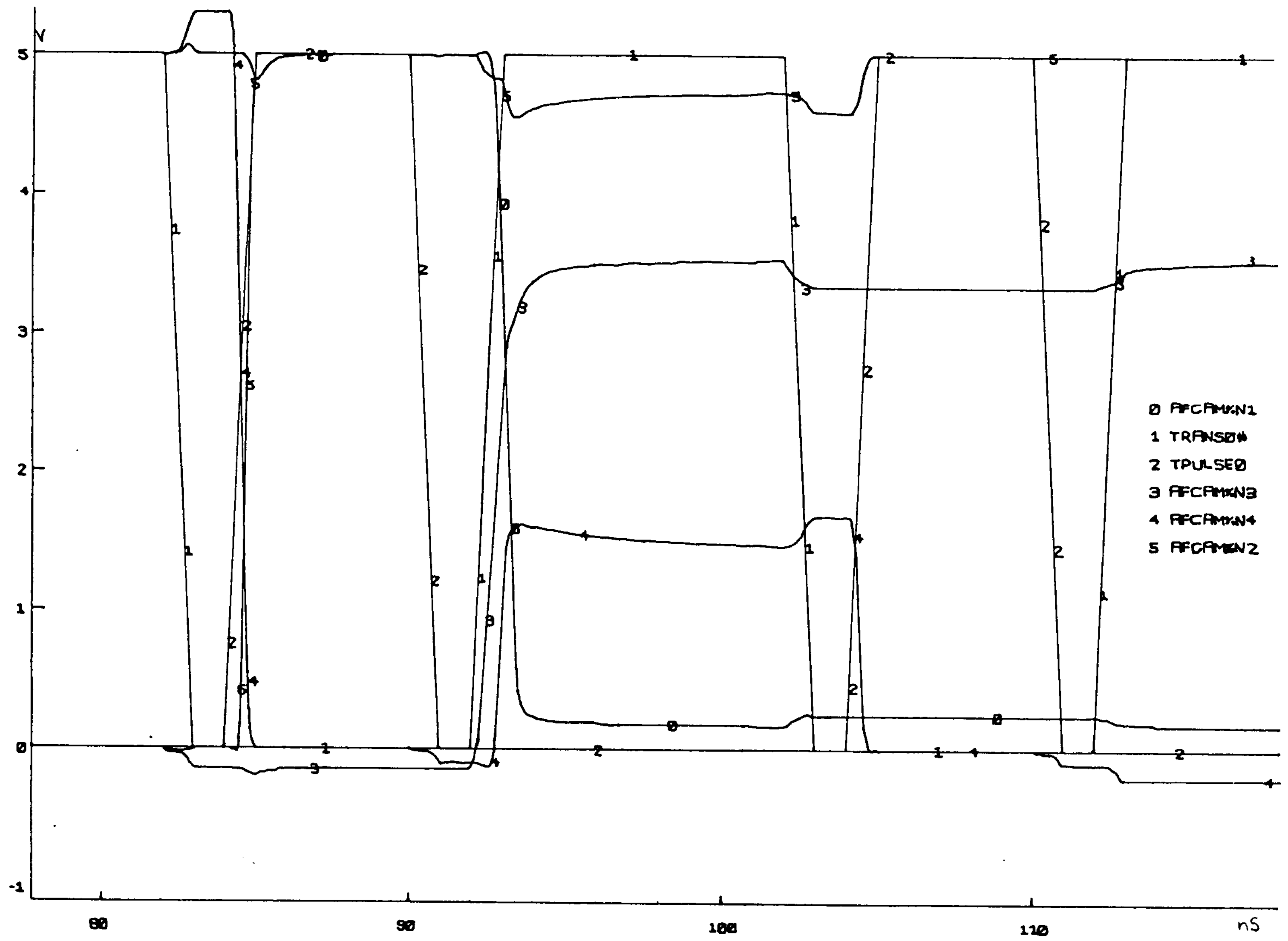


Illustration 26
Shiftdown Simulation, Top Cell

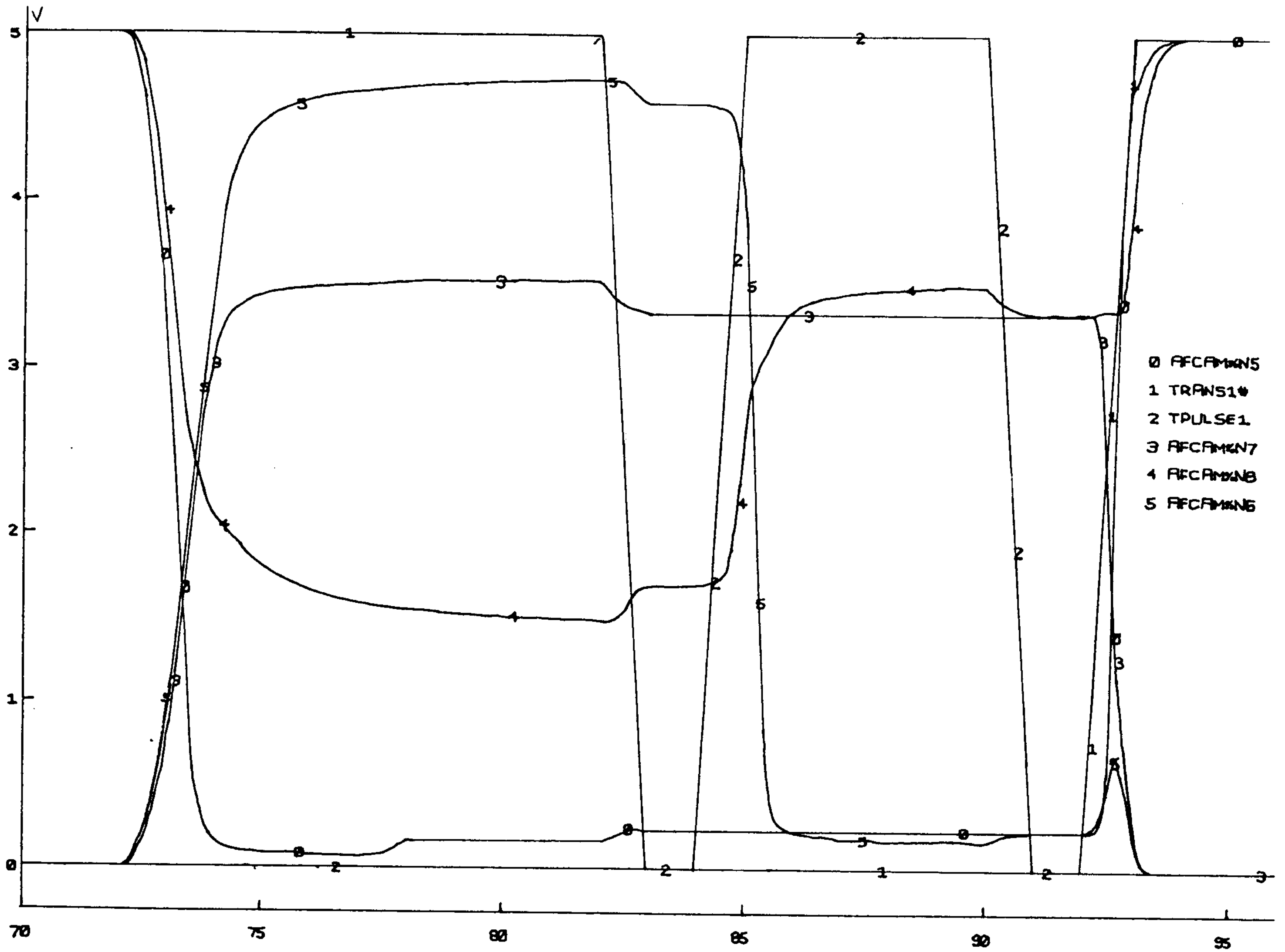


Illustration 27
Shiftdown Simulation, Bottom Cell