Implementation Strategies for a Tagged-Token Data Flow Machine

Computation Structures Group Memo 218
June 1982

Robert A. Iannucci

Table of Contents

List of Figures

Implementation Strategies for a Tagged-Token Data Flow Machine

1. Introduction

The architecture of a tagged-token data flow machine based on the U-interpreter has progressed from an abstract definition of the base language [1, 2], to the definition of the machine's structure [4, 5], to the specification of the instruction set [3]. Work in progress includes the formal functional specification of the machine's behavior [6].

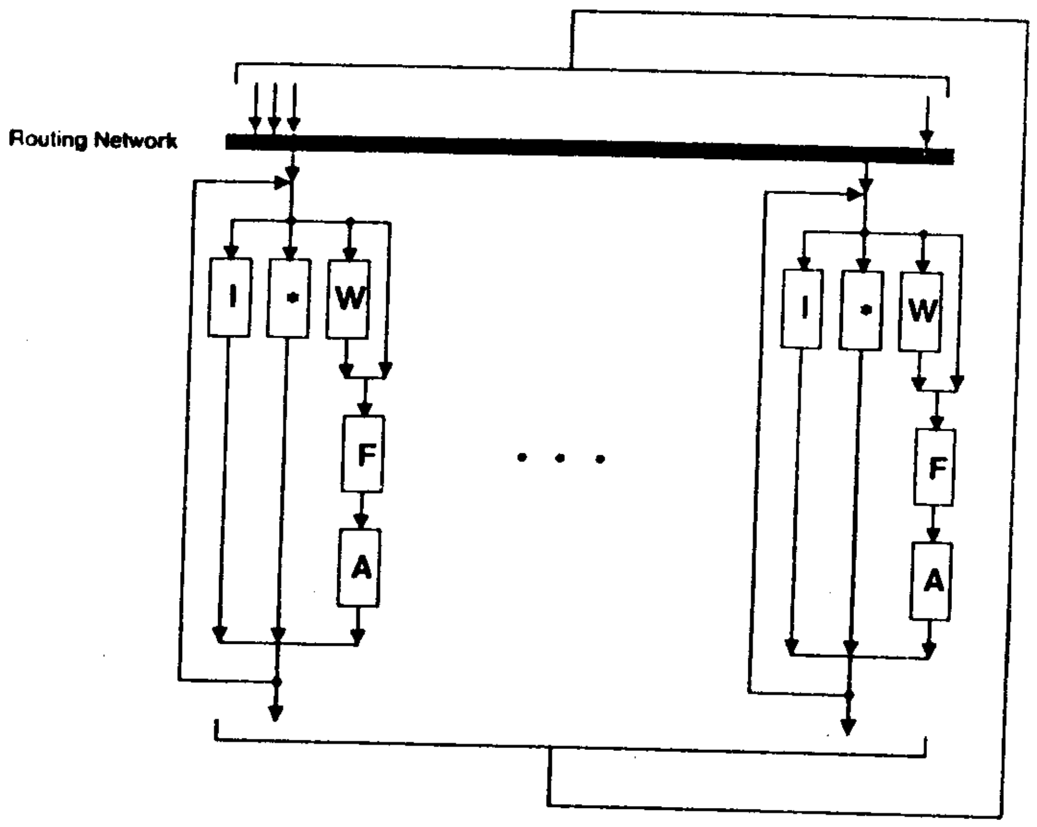
In recent months, the Functional Languages and Architectures group at M.I.T. has reviewed this progress. In so doing, the machine's architecture has been refined, and a clearer idea of its future has emerged. This paper examines three aspects of this review process:

1. **Abstract Architecture and Goals:** A perspective is given for evaluating possible implementations.
2. **Implementation Proposals:** Several of the major proposals are presented.
3. **A Multitasking Micromachine:** One of these proposals is examined in detail (hardware configuration, microprogrammer's interface, etc.).

It should be emphasized that the work presented here represents the efforts of all the members of the group. Specific credit is given where appropriate, however, many of the ideas simply evolved from the discussions.

2. Abstract Architecture and Goals

The abstract architecture of our tagged-token data flow machine is shown in Figure 2-1. Essentially, the machine consists of an n×n routing network connected to n identical Processing Elements (PEs). Each PE contains 1/n of the total program memory in the system, 1/n of the total data (I-structure) memory, a waiting-matching associative memory, an ALU, and simple input/output facilities. Each PE has a connection to the routing network (input and output) as well as a local data path. In essence, each PE is capable of



Key:

W: Waiting-Matching Section

F: Instruction Fetch

A: ALU

I: I-Structure Controller

*: I/O Processor

Figure 2-1: Organization of a Tagged-Token Data Flow Machine

standalone operation without the communication network.

It should be noted that this is not the only possible organization of a tagged-token machine. Figure 2-2 shows an alternative organization which can exhibit markedly different behavior. Rather than organizing the PEs and communication network separately, pieces of the two can be combined.

2.1. Abstract Architecture

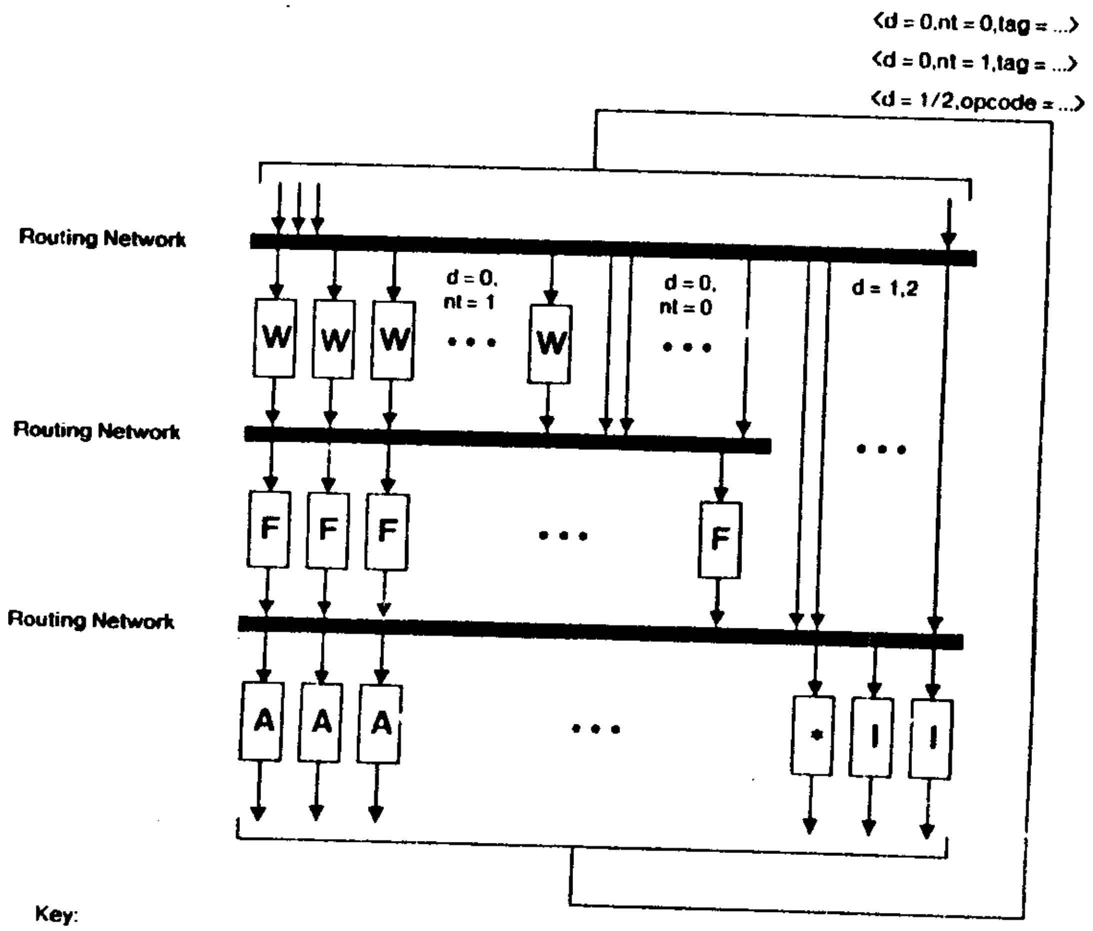
Current work is directed toward the organization of Figure 2-1. As such, we have proposed a PE architecture as shown in Figure 2-3. The PE can be logically divided into three relatively independent subsystems:

1. **d=0 token processing:** This subsystem interprets the data flow graphs proper. It contains the waiting-matching memory for pairing tokens which require partners, a program memory for storing instructions (code blocks), an ALU for performing the operations, and logic for constructing output tokens (tag computation) [3].

2. **d=1 token processing:** The I-structure subsystem processes all requests for fetching data from, and storing data into the I-structure memory. It is capable of handling many requests simultaneously (in that a fetch operation is suspended until the required datum is available). All d=1 tokens contain the tag to be used when the indicated operation is complete. For this reason, no tag computation hardware is necessary.

3. **d=2 token processing:** Many operations may be categorized as *service* or *control* functions. These are carried out by the PE controller which has global access to all of the memories contained in the other subsystems as well as having control over the clocking and error handling services.

All output tokens carry the identifier of the *logical* target PE. Prior to sending these tokens out to the communication network (or routing them back over the local path), the logical PE number is translated into a routing address.



Key:

W Waiting-Matching Section

F: Instruction Fetch

A: ALU

I: I-Structure Controller

*: I/O Processor

Figure 2-2: An Alternative Organization

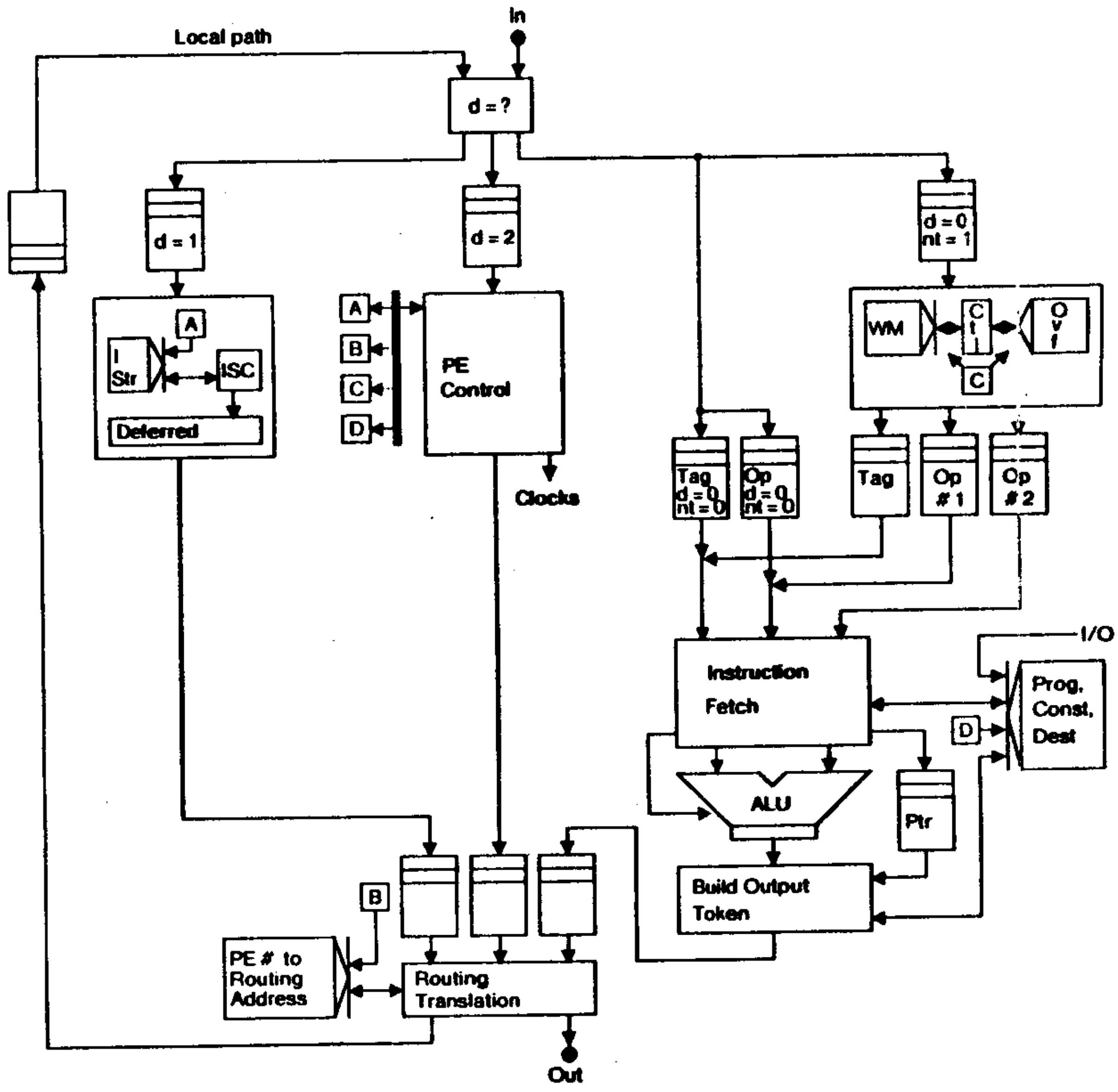


Figure 2-3: Architecture of a PE

2.2. Goals

Each of the possible implementations considered for the abstract machine was weighed against a set of goals. While there was some debate as to the relative importances, the goals were

1. **Adherence to the architecture:** This statement asserts that any proposed design should, in some sense, be isomorphic with the abstract design.

2. **Ability to demonstrate scalability:** Given a program with sufficient parallelism, the machine should show a linear speedup as PEs are added. Further, the machine should not rely on technology in a way that would be difficult to extend.

3. **Lack of artificial constraints:** An important issue is that the basic design should have sufficient capacity (memory, processing speed) to allow for the running of very large programs without needless concern over the physical constraints of the machine. A rough bound on performance would require a PE to be able to execute (on the average) at least 1 data flow instruction every 200 μsec.

4. **Flexibility:** Any machine should be considered as a research tool rather than a "final product". It is foolhardy to presume that we can anticipate all of the questions we may wish to ask about the machine and its behavior beforehand; it is important to keep the design open-ended and the interface to it simple.

5. **Reliability:** While this kind of concern is usually voiced more in an industrial/commercial context, it is of great importance to this project due to its nature. Implementing a single PE without concern for reliability may not present problems, but the desire to have 64 PEs *all working at the same time* cannot be realized using the same philosophy.

6. **Pragmatic concerns:** In view of the fact that we wish to demonstrate a working machine in finite time, the following sub-goals were considered:

   a. **Reasonable cost of PE replication:** Any PE design must live within the constraints of current and projected funding. We have set a limit on the per-PE cost (including necessary support hardware, frames, power supplies, cables, etc.) at $10,000.

   b. **Minimization of external dependencies:** Any effort to extend the project beyond our own research group should be done in a way that the success of the project does not depend totally upon factors beyond our control.

c. **No reliance on constrained resources:** No proposed design should presuppose resources (tools, facilities, people) which cannot be reasonably expected to materialize. This concern is particularly strong in the area of design automation facilities which, in most cases, are themselves research projects.

d. **Ease of construction/testing/debug:** Again, due to the sheer numbers involved, the design should be as simple as possible and as self-diagnosing as possible. Current design practice shows that this can be done fairly easily if considered from the outset of the project.

A rough figure of merit is the cost/performance (C/P) ratio. Cost is measured in $/PE; performance is measured in instructions/sec. (with the understanding that *data flow* instructions are being measured). Therefore, C/P figures quoted in this paper will be understood to have units of ($ sec.)/(PE instructions). Obviously, a "better" design will have a lower C/P ratio than will a "poorer" design.

3. Implementation Proposals

This section describes the six major hardware proposals considered by the group. With each is a brief description of its features and an analysis of how well it addresses the goals. The proposals are presented in roughly chronological order.

3.1. The Discrete VLSI Approach

The desire to implement a PE as a small set of VLSI chips is due to Arvind and Kathail [4].

3.1.1. Proposal

Returning to Figure 2-3, the basic strategy is to partition the design as follows (one VLSI chip per part):

- **Waiting-Matching:** Control logic accepts $d=0$, $nt=1$ tokens and attempts to pair up those with identical tags (using an associative memory). Pineda has designed and fabricated a waiting-matching chip which performs this function for a 64-deep memory. External logic and memory are required to handle overflow, should it occur.

- **Instruction fetch:** As described in [3], d=0 tokens delivered from the waiting-matching section (or directly from the PE input in case of nt=0) denote an instruction in the program memory. The instruction fetch section builds an *operation packet* for the ALU according to a fixed rule.

- **ALU:** This component performs the functions for all d=0 type operations (with the exception of those whose purpose is simply tag alteration, *e.g.*, D, SWITCH, etc.).

- **Token building:** All tag computation is to be done by this subsystem according to the rules in [3]. This involves accepting a partially complete token from the ALU along with the destination list pointer and old tag supplied by the instruction fetch component.

- **I-structure controller:** Tokens of type d=1 are processed here. The operations involve storing and fetching elements from I-structures along with certain low-level storage management primitives. A facility for queueing unsatisfiable fetches is controlled by this subsystem.

- **PE control:** Service type operations (raw read/write operations to the various PE memories, starting/stopping of the clocks, diagnostic operations, etc.) are carried out by a small processor, perhaps a commercially-available microprocessor.

Also, MSI/LSI *glue* (queues, etc.) will be used to bind the chips together. It is estimated that this design could be realized at a per-PE cost of roughly $2000 and a performance on the order of one data flow instruction (on the average) every 10 μsec.

3.1.2. Evaluation

This design clearly adheres very closely to the abstract architecture. Scalability is also easily demonstrated. The detailed design is sufficiently vague that it can be argued that no artificial constraints have been applied. Also, while reliability has not been addressed in detail, it should not seem unreasonable to claim that it could be designed into the chips. In quantities, this proposal has the lowest PE replication cost of the other designs if the actual design can be made to fit into the few VLSI chips outlined. This design has the best C/P ratio of all of those considered (C/P = 0.02).

The proposal is weak in several areas. As in any design with discrete data paths, fixed interfaces, or VLSI technology, the "cost" of flexibility is very high. If anything, the design can be made practical only through the systematic elimination of flexibility. Further, it is yet to be demonstrated that the tools exist to allow a small research group to construct a project of this scale and to get it working in a few years. Several research projects are moving in the direction of specifying a design in a high level language and *compiling to silicon*, but no tools for that purpose exist today.

This proposal is best described as *long-range* and is a good candidate for the ideal "ultimate" project.

3.2. The Migration Strategy

A strategy suggested by Iannucci and Thomas is to construct the tagged-token machine in pieces, with commercially available microprocessors acting to "take up the slack" for components not yet realized in hardware (through emulation). A diagram of such a system is shown in Figure 3-1.

3.2.1. Proposal

As shown in the figure, each major subsystem is housed on a separate card (in this case, an Intel MULTIBUS[1] card). The physical interface between two subsystems consists of a discrete data path which can be logically disabled and replaced with an interface to the MULTIBUS. The philosophy here is that, for subsystems that are not yet implemented, a Motorola 68000 processor attached to the bus can be used to emulate the behavior of the missing subsystem(s). The send-acknowledge protocol will be translated into a MULTIBUS interrupt handshake sequence (in essence providing a send-acknowledge protocol between a hardware component and a software module).

Cards implementing the various subsystems in hardware can be staged into the design as manpower permits. Top-card cables would provide the discrete data paths between hardware components. This strategy has the additional benefit of being able to isolate the

[1]Trademark of Intel Corp.

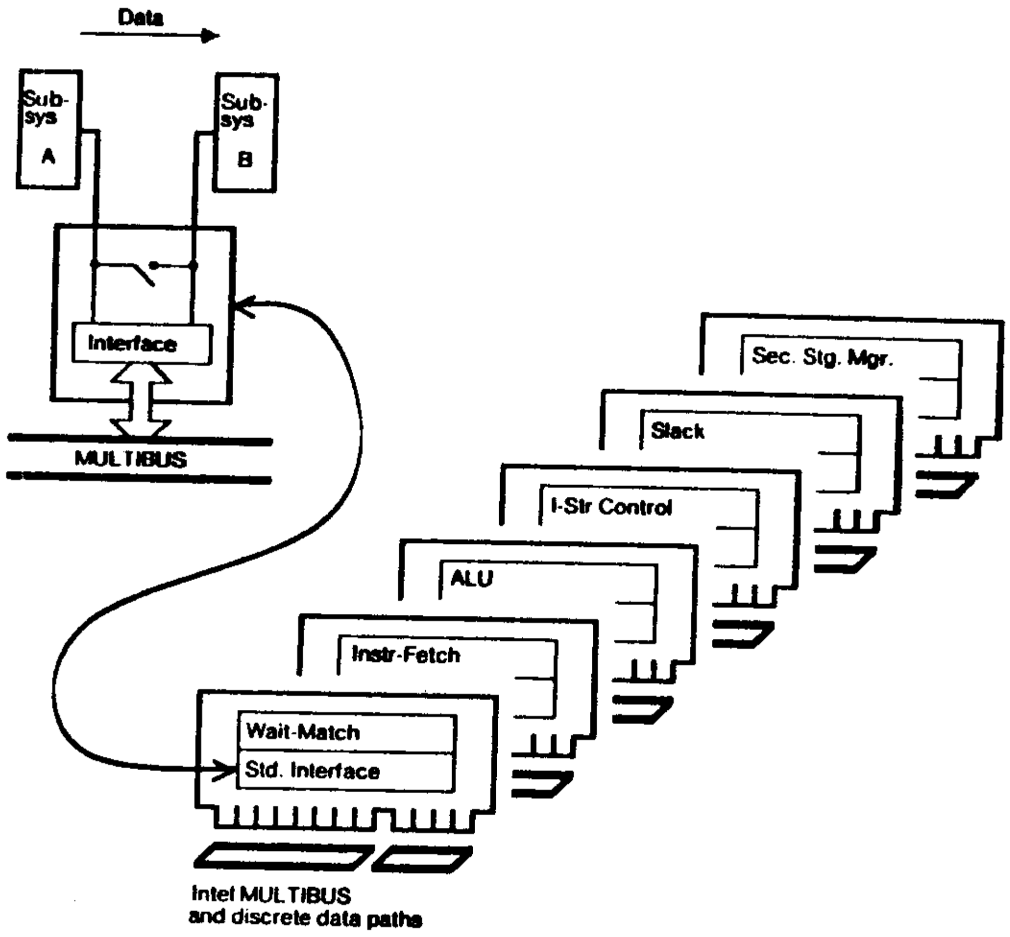


Figure 3-1: Multi-Card PE Migration Strategy

hardware modules even *after* they are designed and debugged - for testing and diagnostic purposes.

The per-PE cost of this design is close to $15,000. In final form (all discrete data paths implemented), it would have the same performance as the VLSI design - 1 instruction every 10 μsec.

3.2.2. Evaluation

This design addresses most of the goals quite effectively; due to the fact that the ultimate version of the PE implements the subsystems and data paths as the architecture indicates, the system matches the architecture well. This design is more flexible than the VLSI version in that hardware modules can be staged in and out simply and easily without major impact (allowing for simple design changes). As in the VLSI machine, reliability would have to be considered more carefully. This design also has a very good C/P figure of 0.15.

No dependence exists on technologies far beyond the state of the art in this design. This, along with the well-defined interfacing strategies, allows the project to be partitioned among as many (or as few) designers as are available. As has been pointed out, this design exhibits excellent debuggability and testability characteristics.

The single biggest drawback to this strategy is the cost per PE. As shown in the diagram, each PE would be 6 (perhaps 7) MULTIBUS cards. The estimated cost exceeds the $10,000 per PE threshold defined by the goals.

3.3. The Motorola 68000 Based Single-Card PE

Arvind and Thomas suggested a design based on a commercially available single-card computer (see Figure 3-2).

3.3.1. Proposal

In this design, no time or expense is incurred in designing PE hardware; the philosophy is to interconnect 64 commercially-available MULTIBUS processor cards (M68000 based by virtue of the available software). Each card would emulate (via multitasking) or

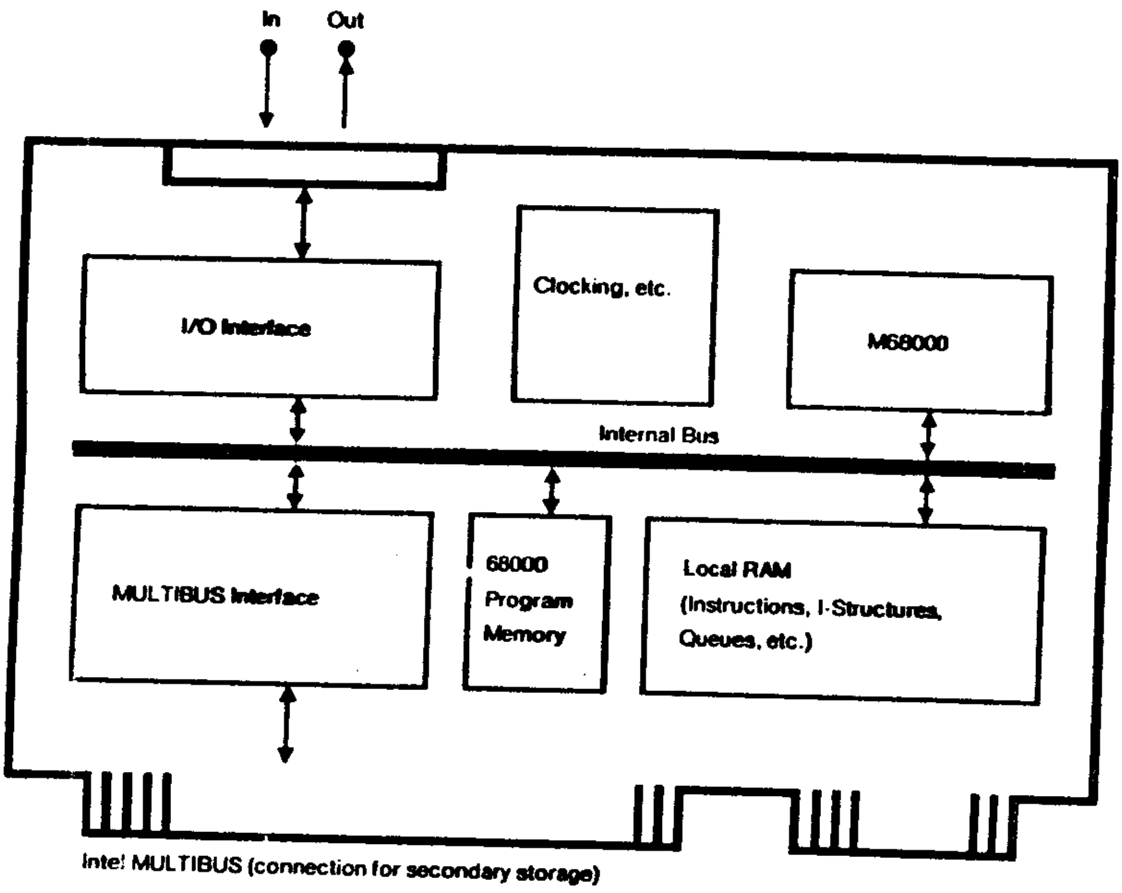

In
Out
I/O Interface
Clocking, etc.
M68000
Internal Bus
MULTIBUS Interface
68000 Program Memory
Local RAM (Instructions, I-Structures, Queues, etc.)
Intel MULTIBUS (connection for secondary storage)

Figure 3-2: Motorola 68000 Based Single-Card Computer PE

simulate the behavior of one PE.

An optimistic estimate claims that it would be possible to construct a 64 processor machine at a per-PE cost of $2500. Performance would be on the order of one instruction every 150 μsec.

3.3.2. Evaluation

In support of this approach, it is by far the least manpower-intensive, it offers the ultimate in flexibility, and the cost of replication is low. C/P is good but not excellent at 0.375.

This proposal has several drawbacks. Primarily, the adherence to the architecture is harder to argue in the case of emulation; if a simulation is chosen instead (to preserve the internal as well as external characteristics of the abstract PE), performance would suffer considerably (thus artificially constraining the types of programs which could be executed by virtue of slowness). Reliability of the single-card computers is also questionable in that none are available with error-corrected memories.

3.4. The Motorola 68701 Based Single-Chip Computer PE

The author also proposed a discrete data path machine which relied less on external factors than the discrete VLSI approach yet, at the conceptual level, adheres closely to the architecture.

3.4.1. Proposal

The strategy behind this design was to use off-the-shelf single chip microcomputers (random access memory, ultraviolet erasable programmable read-only memory, input/output, timing/interrupt facilities, and a CPU on a single chip) like the Intel 8748 or the Motorola 68701. One microcomputer would be used to implement each major subsystem (see Figure 3-3). Further, to reduce the amount of unnecessary data movement, a global heap storage would be used to contain the token data; the discrete data paths would be used to pass pointers among the subsystems.

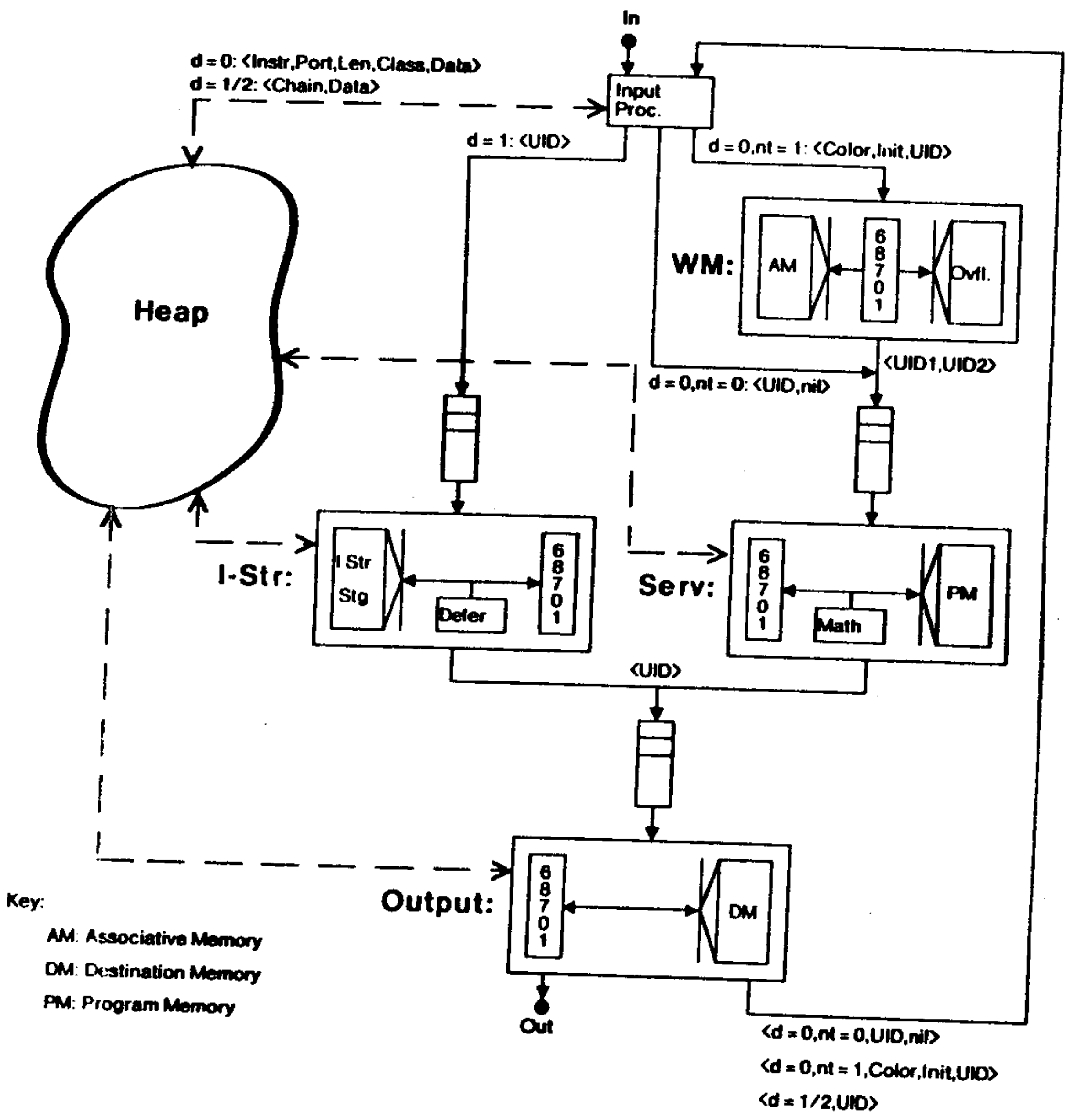


Figure 3-3: Motorola 68701 Based PE

As a token arrives from the network, an input processor would send the bulk of the data (see Figure) to the heap; the heap manager would return a unique identifier which would be used by the other subsystems to access the data. The heap would be a contention-free four port memory (using bipolar components, the cycle time of the heap is approximately five times the fastest single instruction time of the 68701).

Each of the other subsystems (with the exception of waiting-matching) would have access to the heap. Also, each subsystem would have some local memory for non-shared data (*e.g.*, program memory, I-structure storage, etc.). As the subsystems ran independently (no memory conflicts, even at the heap), the execution rate would be determined by the pipeline step time. For the average case, the ALU/Service Section could execute a data flow instruction in approximately 150 μsec., relying on an Intel 8087 for floating-point arithmetic.

A rough partitioning showed that this design would span roughly three MULTIBUS sized cards, and the per-PE cost would be nearly $6000 (including frames, power supplies, etc.).

3.4.2. Evaluation

As pointed out, the advantages of this design derive from its adherence to the organization and partitioning of the abstract machine. It is fairly flexible in that the control programs for each 68701 can be readily changed, although the physical paths are somewhat constraining. The design does not rely at all on the availability of advanced design tools or facilities that are not readily available.

The disadvantages, however, are equally noteworthy. The design relies on the relative speeds of the heap *vs.* the 68701 to assure interference-free behavior. This kind of constraint is usually difficult to preserve as individual technologies (single chip computers, memories) advance. The raw speed is not exceptional; moreover, the cost/performance is the worst of all the designs considered (C/P = 0.9). Reliability would also have to be addressed.

3.5. The 3-Microprocessor Shared Memory PE Structure

Pineda took a slightly different approach; by using a contention-based shared memory structure with off-the-shelf components, a physically simpler design could be realized.

3.5.1. Proposal

Referring to Figure 3-4, we see that the design is partitioned into four major subsystems; three processors and one shared memory (bus-connected to the processors). The paths shown in the diagram are logical, not physical. Three physical processors allow for some PE-level concurrency (missing in the single-card computer scheme), and the design allows for the use of commercially-available math coprocessors (like the Intel 8087). This design, like the 68701 based PE, takes advantage of pointer movement rather than data movement. Unlike the 68701 scheme, the three processors communicate the pointers through shared memory and synchronizing semaphores.

Pineda's analysis shows data flow instructions can be processed at a rate of about one per 250 μsec. The per-PE cost is in the neighborhood of \$2500 (packaged as a single logic card).

3.5.2. Evaluation

This design is very attractive in terms of cost, manpower, flexibility, lack of dependence on external factors and general simplicity.

A primary disadvantage is the machine's instruction execution speed. Note, however, that while this design is somewhat slower than the 68701 based machine, it does have a better cost/performance figure (C/P = 0.625). However, it does not exhibit an obvious isomorphism to the abstract machine.

3.6. The AMD 2903 Data Flow Multitasking Micromachine

Early on in the evaluation process, the author proposed a multitasking micromachine as an implementation strategy. This idea was dismissed and then later resurrected.

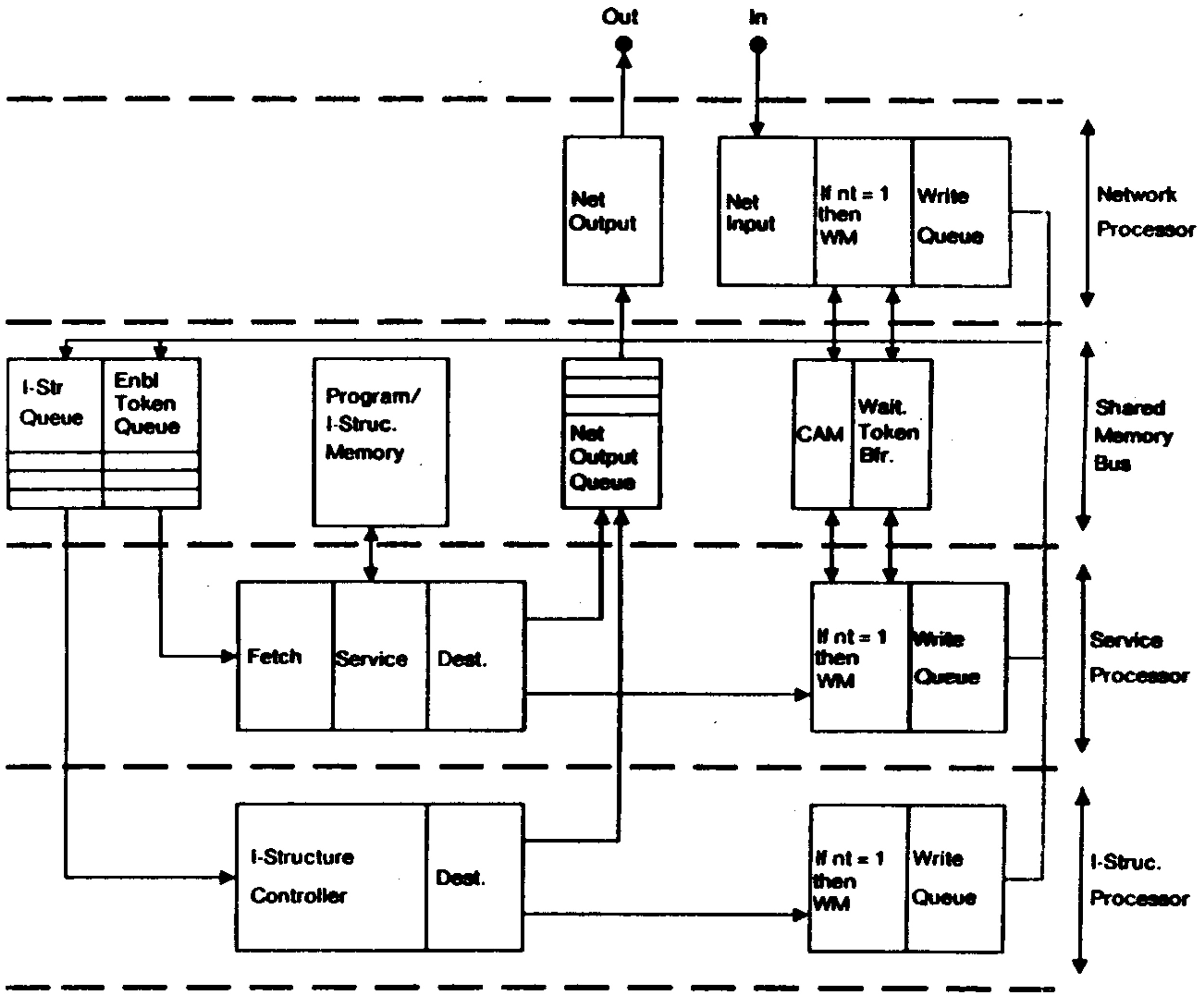


Figure 3-4: 3 Microprocessor Shared Memory PE

3.6.1. Proposal

The philosophy of this proposal was to design a machine that could be readily realized with current components and techniques which in and of itself exhibited substantial capacity (processing speed and memory size), comparable to a MOS VLSI implementation. The machine itself should be easily constructed, debugged, and tested; moreover, it should have better reliability than commonly-available single card systems. It should be open-ended to the extent that, not only should it itself be able to interpret data flow graphs at nontrivial rates, but it should allow for the staging of the design from code-based modules to hardware-based modules (as does the multi-card migration strategy). Further, it should be inexpensive to replicate.

A machine which meets these criteria is shown in Figure 3-5. A detailed description of its operation is given in a later section. Its low-level structure relies on microcode level context switching in the spirit of the Xerox Alto [9] and Dorado [7]. As in the M68000 single-card design, each subsystem in the abstract machine is represented as a software (microcode) module. Unlike the single-card system, context switching is extremely efficient because of explicit hardware support for it.

The data paths, memory, and arithmetic elements are sufficiently wide (32 bits) that no convoluted code need be constructed to manipulate normal data objects (*e.g.*, floating-point numbers, memory addresses). Automatic alignment circuitry and length/boundary hardware eliminate the vagaries of manipulating objects smaller than 32 bits as well.

The control storage subsystem implements a very sophisticated yet simple branching mechanism to allow for 2-way, 4-way, 8-way, ... 256-way branching on any data object. Multiway branching can be performed on contiguous and noncontiguous bits within the same eight bit byte.

The maintenance subsystem contains a microprocessor which is separate from the main data paths; it facilitates initialization, testing, debug, and control of the micromachine. To further enhance the reliability of the primary memory (constructed from 64K bit or 256K bit dynamic RAMs), it is error-corrected.

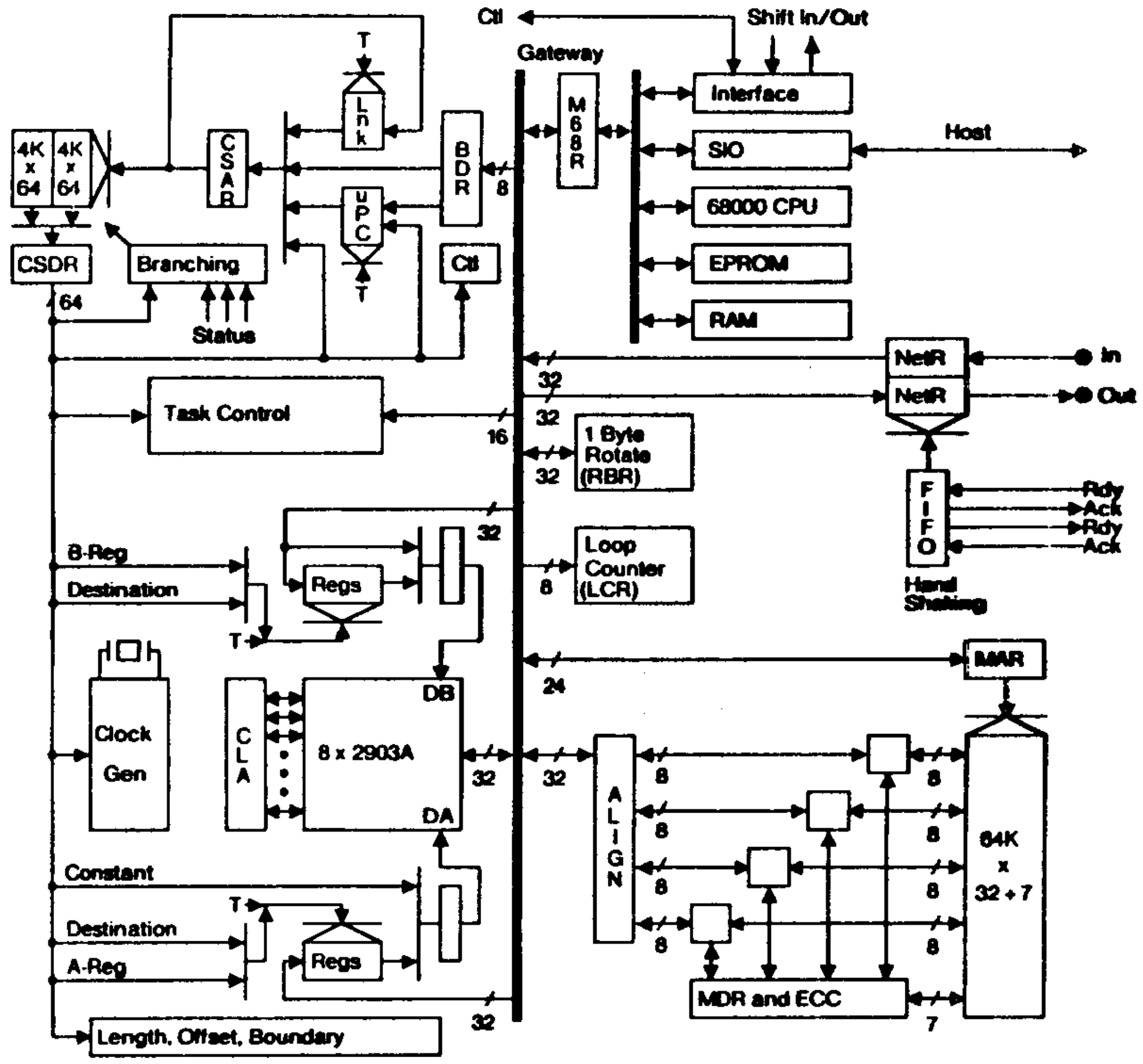


Figure 3-5: AMD 2903 Based Data Flow Micromachine

The rough per-PE cost is \$3500; the data-flow execution rate is in the neighborhood of one instruction per 20 μsec.

3.6.2. Evaluation

This proposal has one of the best cost/performance figures of all the designs examined (C/P = 0.07), second only to the VLSI machine. It is flexible in that the control logic is implemented in microcode rather than in silicon. It is also the only design to specifically address the reliability issue. Debugging is significantly enhanced through the use of a maintenance processor for which software already exists. Since it is constructed from off-the-shelf components, it relies only on design tools for card-level wiring. The design makes no outlandish claims on the underlying hardware; hence the micromachine should scale with technology.

The primary disadvantage is the difficulty of arguing adherence to the architecture. To do this properly, it will be necessary to demonstrate that the external characteristics of the abstract PE have been preserved. The internal behavior relies on the structure of the microcode; strict interfaces must be maintained between the microcode tasks (analogous to the "strict" wire interfaces between subsystems in the abstract machine).

4. A Multitasking Micromachine

4.1. Basis and Structure

As described in the previous section, the AMD 2903 based micromachine of Figure 3-5 was conceived as a general research tool, and answers well most of the objections raised in evaluating the other machines. This section reviews the major subsystems of the micromachine, and describes the microinstruction format.

It is important to recall the salient features of this design while reading the detailed descriptions; they are presented here in summary:

- **Hardware-supported context switching:** Tasking on a conventional machine is generally limited by the overhead in performing a context switch. The biggest problem is generally in saving the programmer-visible machine state (addressing and data registers, status flags, etc.). The Xerox Alto and Dorado [7, 9] solve this problem at the microcode level by *replicating* machine state wherever it occurs, and by putting a fixed limit on the number of tasks which can concurrently execute. As these machines are register-based, this means that each task has its own set of registers. Thus, rather than rolling out $task_i$'s registers and rolling in $task_j$'s registers (lots of memory references), the physical hardware is simply replicated. Switching then becomes trivial.

Unfortunately, at the microcode level, the machine state manifests itself in many more ways than just the general-purpose registers and the program counter. Any specific hardware resources (*e.g.*, loop counters, MARs, MDRs, branching registers, etc.) must also be duplicated. The trend exhibited by Xerox in the Alto-to-Dorado transition was to increase the machine complexity by doing just this.

In the design of our micromachine, this was the original thinking. However, it became apparent shortly thereafter that it does not make much sense to duplicate *all* such machine state for the following reason: in that the tasks for the data flow micromachine will be communicating with each other rather heavily (unlike the Alto and Dorado where the tasks are rather autonomous), it will be necessary to implement an atomicity mechanism for using shared storage (a common situation). Memory-based synchronization flags would represent a serious performance degradation (due to the relative slowness of the memory). Rather, it is better to allow the microprogrammer some explicit control over the decision to switch tasks; in an "atomic" section of code, the task switching would be temporarily disabled (*e.g.*, for the duration of a memory operation). Giving the microprogrammer fine-grained control over the decision to enable task switching allows us to further reduce the hardware complexity by making *all use* of non-duplicated resources atomic. In the case of MAR and MDR, this actually comes for free (since the memory reference must be atomic, anyway).

- **Adequate capacity:** The micromachine is designed to contain a minimum of 256K 8-bit bytes of error-corrected dynamic memory (refreshed by a microcode task). By replacing the 64K bit chips with pin-compatible 256K chips, the per-PE memory size is 1 Megabyte, fully error-corrected. The control storage size is 8K×64 bits. The clock period is under microprogram control; register-to-register arithmetic will cycle in under 200 nanoseconds.

- **Efficient branching:** The control storage is two-way interleaved; each access fetches two consecutive microinstructions. Late in the cycle, a boolean value will select the odd- or the even-addressed microword. This allows conditional branching to occur without the typical one-cycle pipeline delay penalty which most machines exhibit. This results in microcode which is easier to write without any loss of performance.

Further, efficient multiway branching on any number of bits in an arbitrary 8-bit byte can occur in a single cycle (in parallel with arithmetic *and* main memory operation). Each task has its own micro program counter, and has the capability for one level of subroutine linkage. Task switching occurs without the loss of any microcycles.

- **Reliable operation:** The micromachine, as noted, uses error-corrected memory. Several hardware checks are also performed (parity on all control storage fetches). All errors cause a special error-handling task to be enabled. It responds by communicating the nature of the error to the maintenance processor which can perform further logout and analysis. The maintenance processor has *full access* to the micromachine's state (including the contents of the control storage, CSAR, and CSDR). Further, a "diagnostics" task is enabled to run when no other work is being performed by the micromachine (background operation).

4.2. Major Subsystems

The micromachine is partitioned into four major subsystems; the ALU, the control (microprogram) storage and sequencer, the main storage, and the maintenance subsystem. These four systems communicate with one another through

- a 32 bit wide shared data bus,
- a 64 bit wide microinstruction bus, and
- a status bus.

During any microinstruction cycle, any or all of the following may take place:

- **Source selection:** The microprogrammer may select two general-purpose registers (GPRs), a constant and a GPR, a constant and a data bus-connected register, or a GPR and a bus-connected register as inputs to the ALU.

- **Destination selection:** The microprogrammer may select one or more targets for the result of a computation: a GPR, main memory, or a bus-connected register. The offsets/boundaries are under microprogram control.

- **Branch selection:** Many options are available for performing multiway branching; most can be overlapped with other useful operations.

- **Memory reference:** The microprogrammer has low-level control over the behavior of the dynamic memory subsystem. This allows for simpler hardware; refresh is handled by a microcode task. The details of manipulating the memory are largely handled by a set of fairly powerful macro operations for the user who does not care about such things.

- **Status setting:** Many status conditions can be latched or used in the conditional branching hardware directly. A hardware-managed loop counter is provided for writing single microinstructions which can loop on themselves while automatically decrementing and testing a count.

- **Next task computation:** Hardware examines the pending requests for all of the 16 possible tasks and selects the next task. If the microinstruction specifically allows task switching, the current micro program counter will be saved, and the new task's microprogram counter will be fetched. No delay cycle is necessary in order to do the task switching.

4.2.1. ALU

The arithmetic subsystem is designed around a set of eight AMD 2903A bipolar bit-slice processor chips. External lookahead carry logic is also provided. The register file internal to the 2903s is ignored, and an external file of 16 groups of 32 32-bit registers is used instead (one group of registers per task). The register file is actually duplicated (one copy for the ALU A-input, the other for the B-input). Writing into the files is done to the same address in each, but reading may be done using two different addresses. This gives, in effect, a fast dual-ported memory.

Multiplexers and control logic allow the ALU A-input to be driven from one register file or the microprogram-specified constant, while the ALU B-input can be driven from the other register file or the data bus. Alternation of the register addresses between source and destination, as well as the padding-out of the addresses with the current task ID, is also handled by the hardware.

4.2.2. Control Storage

The primary function of the control storage subsystem is to fetch microinstructions from the microprogram memory in some meaningful sequence. This machine does not employ any hardware for performing arithmetic (*e.g.*, add 1) on the microinstruction addresses. Instead, each microinstruction contains enough information so that the next microprogram address (held in the μPC) can be computed by fairly simple bit-substitution.

There are several hardware-supported microprogram address formation modes:

- **Jump:** This is the normal mode of operation. Bits in the microinstruction are substituted as-is into the μPC. Thus, unconditional control transfer and normal sequential flow are handled in the same manner.

- **Jump to subroutine:** The current μPC is saved in a special memory, and the bits in the microinstruction are interpreted as the new μPC (the starting address of the subroutine) as in the *Jump* case. The hardware only supports one level of subroutine call.

- **Return from subroutine:** The μPC is restored, and the low order bit is forced on. This simplification implies that all *Jump to subroutine* microinstructions must be mapped to even-numbered addresses in the microprogram memory. Due to the fact that each microinstruction in a sequential flow can be mapped to *any* location equally well, this is not a significant problem.

- **Multiway branch:** The eight low bits of the microinstruction-specified address are interpreted as a *Mask*. The remainder of the bits are substituted directly into the addressing register as in the *Jump* case.

  The hardware contains a special 8-bit register (the Branch Data Register, or BDR) which is connected to the data bus. It is a legal destination target, and can thus be set under microprogram control to any value.

  The Mask and BDR are combined to form the low-order eight bits of the μPC by the following rule. For $0 \leq i \leq 7$, examine Mask_i.

  * **if 0:** μPC_i is left unaltered.

  * **if 1:** μPC_i is set to the value of BDR_i.

  In other words, bit positions in the mask equal to zero correspond to *don't care* bits in the BDR.

By this mechanism, many types of multiway branching can be performed. Simple 2-way branching is done whenever there is exactly one "1" bit in the Mask. Likewise, 4-way branching corresponds to two "1" bits, and so on. The limiting case is 256-way branching with all bits in the Mask (8 of them) as "1".

Two-way branching on hardware status conditions is handled by a separate mechanism. In the above, branching is performed by computing a new μPC from the old μPC, a Mask, and an arbitrary 8-bit quantity. More frequently, the microprogrammer will want to branch based on some boolean *status* condition. Traditionally, this is done by selecting the condition (ALU overflow, etc.) and using it as one of the address signals for the microprogram memory.

We adopt a similar scheme here, but implement it in a way that serves to put fewer constraints on the microprogrammer. In the traditional approach, using the status bit as an address line meant that the status condition had to be available early in the microcycle (since the availability of the next microinstruction depends on accessing the control storage, and accessing control storage depends on the status bit). If the bit is to be *developed* during the current microcycle, as would be the case in any arithmetic status, the microprogrammer must do one of two things:

1. Extend the microcycle to accommodate the total required delay. This is likely to be nearly twice the normal cycle time, slowing down the overall performance accordingly.

2. Set the status condition one cycle ahead of time (*i.e.*, do the operation which sets the bit, execute another unrelated cycle, then perform the branch). This type of microcode is typically very hard to write, debug, and maintain.

To avoid this problem, we adopt the strategy of fetching *two* microinstructions per cycle as in a two-way interleaved memory. The fetched instructions form an even/odd pair (addresses i and $i+1$, where i is even). Conditional branching is done by selecting, late in the cycle, one of these two microinstructions. In effect, we are still using the status bit as the low-order address line; the primary difference is that the settling of this line goes on *in* *parallel* with the array access rather than *in serial.*

The microprogrammer is given the ability to select among a number of status conditions. A field in each microinstruction is reserved for this purpose. When decoded, it identifies one of these possible conditions which then, in effect, becomes the low-order address bit. Note that, in the case of non-conditional instruction processing, we still need some ability to indicate *fixed* branching. Two special decodes denote the always-even and always-odd cases.

4.2.3. Main Storage

The main storage is configurable as 256K or 1M 8-bit bytes (single error corrected/double error detected - SEC/DED) organized as 64K or 256K 32-bit words. Several basic cycle types are possible:

- **Read:** Initiated whenever the address register (MAR) is set; error-corrected data are available in the data register (MDR) after the fixed access time.

- **Write:** A full write cycle requires setting of the MAR and MDR. The error-correction circuitry will compute check bits for the MDR data, and the write will be initiated.

- **Partial write:** Writing of less than 32 bits requires prefetching (Read/Modify/Write). The prefetch operation is under microprogrammer control (using an explicit *Read* cycle). Modification of less than 32 bits of the fetched data is assisted by a data alignment network which connects the MDR to the system bus. This permits the modification of 8-bit items on 8-bit boundaries and 16-bit items on 16-bit boundaries. Again, the error-correction circuitry computes the new check bits. The length information is specified in the microinstruction; the offset information may be specified by the microinstruction or by the low 2 bits of the MAR.

- **Refresh:** Similar to a Read cycle, it is initiated by setting the MAR. Only the Row Address Select lines (RAS) to the memory are asserted. A *Refresh* cycle is shorter than a normal *Read* cycle.

4.2.4. Maintenance Subsystem

The maintenance subsystem is designed to deal with the issues of debugging, testing, initialization, error-handling, and host input/output. It is made up of a commercially-available microprocessor, volatile and non-volatile memory, input/output interfaces, timing facilities, interrupt handling logic, and a gateway to the primary micromachine bus.

At power-on, an initialization program in the non-volatile memory will initialize the maintenance subsystem and the micromachine (control storage, main memory, registers, clocking). It will also attempt to establish communication with a host computer (file service, etc.).

4.3. Microinstruction Format

The microinstruction is made up of many orthogonal fields, some of which are encoded. They are described here in broad outline only.

- **A-Source:** Selects a 16-bit immediate constant or one of the 32 GPRs as the input to the A-side of the ALU.

- **B-Source:** Selects one of the 32 GPRs or one of the 32 bus sources as the input to the B-side of the ALU.

- **Destination:** Selects one of the 32 GPRs or one of the 32 bus sinks as the target for the current operation.

- **Write-enable:** Allows for concurrent writing into several destinations (special cases that occur frequently). These fall into three categories:

  1. Register/Bus sink

  2. BDR

  3. MDR

  Any combination of these three groups is permitted.

- **Length/Offset:** The operand length (2 bits) and offset (2 bits) may be explicitly specified. These fields play an essential role in manipulating memory for items less than 32 bits in length. Length/offset resolution is to the byte (8-bit) level.

- **ALU function:** Controls the operation of the AMD 2903As.

- **Hardware scheduler command:** Enables or disables task switching, queries the current tasking status, requests or masks a particular task, etc.

- **Cycle length:** Controls the clock generator by adding time (in increments of 50 nanoseconds) to the basic cycle. Useful for allowing the machine to operate at a cycle length appropriate for a particular operation; it eliminates the need to run the machine always at the "slowest" cycle speed.

- **Odd/even select:** Allows the specification of the branch condition.

- **NA/Constant/Mask:** Specifies one of the following:

  1. Next microinstruction address

  2. Mask for computing the next address

  3. A 16 bit arithmetic or logical constant

4.4. The Microassembler

Thomas [8] has built a flexible, general-purpose macro assembler which will be adapted for use with the micromachine. Microinstruction syntax is equational, with some limited use of keywords for branching and other controls. Definition of the syntax is beyond the scope of this paper and will appear separately.

One of the major issues in the design of such a microassembler is the problem of assigning microinstructions to control storage addresses. Due to the regular nature of the microinstruction addressing (we have not used a blocked control storage structure), this is a containable problem. Assignment can be done quite effectively as a three step process:

1. Select cells for the targets of all multiway branches starting with 256-way, then 128-way, then ...

2. Select cells for all fixed odd/even pairs:

   - 2-way branching using the odd/even select mechanism

   - Subroutine linkages

- Microinstructions using immediate constants and their successors

3. Assign the remaining microinstructions freely to the remaining cells.

5. Conclusion

This paper has attempted to provide an overview of the thought processes that led our group to the current projects. We reviewed six major proposals and evaluated each one according to a set of fixed goals.

One of the proposals, a multitasking micromachine, was examined in great detail. It is our intent to construct this processor as a single logic card and to replicate it (64 copies) for the purpose of building a large data flow research vehicle.

Our preliminary goals call for the following milestones:

- **Design:** completion of the design (to the point that a wire-wrap list and a parts list have been generated) by August 31, 1982

- **First machine:** by December 31, 1982

- **First version of the microcode:** by March 31, 1982

- **64 machines:** by October 31, 1983

- **Full operation:** by December 31, 1983

References

1. Arvind, and Gostelow, K. P. The U-interpreter. *COMPUTER* (December 1981). To appear.

2. Arvind, K. P. Gostelow, and W. Plouffe. An Asynchronous Programming Language and Computing Machine. Tech. Rep. 114a, Department of Information and Computer Science, University of Californiav, Irvine, California, December, 1978.

3. Arvind, and R. A. Iannucci. Instruction Set Definition for a Tagged-Token Data Flow Machine. Memo 212, Computation Structures Group, Laboratory for Computer Science, MIT, Cambridge, Mass., December, 1981.

4. Arvind, and V. Kathail. A Multiple Processor Dataflow Machine That Supports Generalized Procedures. Memo 205, Computation Structures Group, Laboratory for Computer Science, MIT, Cambridge, Mass., February, 1981.

5. Arvind, V. Kathail, and K. Pingali. A Dataflow Architecture with Tagged Tokens. Tech. Rep. TM-174, Laboratory for Computer Science, MIT, Cambridge, Mass., September. 1980.

6. Iannucci, R. A., and J. Pineda. Functional Specification for a Tagged-Token Data *Part 0: System Overview and Processing Element Structure.* Memo unpublished, Functional Languages and Architectures Group, Laboratory for Computer Science, MIT, Cambridge, Mass., March, 1982.

7. Lampson, B. W., and K. A. Pier. A Processor for a High-Performance Personal Computer. Xerox Palo Alto Research Center, January, 1981.

8. Thomas, R. E. A68K Macro Assembler User's Guide. Functional Languages and Architectures Group, Laboratory for Computer Science, MIT, Cambridge, Mass., November, 1981.

9. *ALTO: A Personal Computer System - Hardware Manual.* Xerox Palo Alto Research Center, Palo Alto, California, 94304, 1979.