An Engineering Implementation of the TTDA

Computation Structures Group Memo 270

February 12, 1987

Gregory M. Papadopoulos

This report describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-84-K-0099.

Table of Contents

List of Figures

An Engineering Implementation of the TTDA

Gregory M. Papadopoulos
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

1. Overview

This document provides the preliminary design for a dataflow multiprocessor prototype based on the M.I.T. Tagged Token Dataflow Architecture (TTDA). The goal is a practical experimental vehicle consisting of 256 processing elements and structure controllers and a suitable multistage communication network to yield an overall performance in excess of 1000 MDIPS.[1] We expect the facility, used in conjunction with the Id programming language, to become the first truly *general-purpose* multiprocessor which will significantly out-perform current von Neumann supercomputers on a broad range of applications. By using relatively conservative TTL and CMOS gate array technology, a complete prototype array should be available by January 1990.

Meeting this ambitious goal requires extraordinary attention to design risk management. We therefore use the "pure" dataflow model provided by the TTDA due to our experience with the architecture and in spite of recent creative ideas of hybridizing a von Neumann machine [Iannucci].[2] Furthermore, there is limited opportunity for optimization. For example, the current compiler and instruction set yields a dynamic inefficiency of about 2:1 as compared to optimized FORTRAN code on a sequential machine. For the purposes of the prototype, this inefficiency is tolerable.[3]

The TTDA is not *itself* a specification for a practical hardware design. If the prototype is to form a basis for future competitive parallel machines, then the prototype implementation must address the possibility of good cost/performance. We are critical of mechanisms that might lead to an expedient prototype but do not lay a sound foundation for future work. For example, a bit-sliced microcoded processing element would be relatively easy to implement and provide a large degree of experimental flexibility. However, it is still an *interpreter* for the TTDA. It lacks an efficient

[1]MDIPS = Million Dataflow Instructions Per Second.

[2]PhD dissertation in preparation.

[3]We certainly hope and expect that further improvements in compiler technology will gain some of this back, but little effort will be made at the hardware level.

implementation of data synchronization that we claim is fundamentally important and is unable to exploit the clean pipeline properties of data-driven evaluation. We search for a design with the following properties:

- **Technological Scalability.** As the technology is scaled to ECL and GAAS, the cycle time of the machine should decrease in direct relationship to improved propagation delays.
- **Pipelined Architecture.** In the RISC tradition, a simple non-blocking pipeline. Single cycle throughput. No complex instructions.
- **Completeness.** The processor state can be effectively manipulated by the dataflow instruction set. There is no "controller" microprocessor associated with each node.

The preliminary design to be presented largely meets these objectives and should provide an efficient yet relatively flexible research vehicle. The highlights are:

- Explicit token storage yielding a three-stage pipeline waiting-matching section that sustains 10 million wait/extract operations per second.
- Single hardware design for both processing elements and structure controllers, unifying tags and pointers.
- Fixed 64 bit data size with 8 bit type. Fixed size instructions, unary and binary, with at most two destinations.
- Two megawords of token/instruction storage per module (DRAM). Token and instruction caches.
- Seven-stage pipelined TTL/CMOS Gate Array Design, 100ns. instruction cycle time.
- Multistage packet switching network, 256 ports at 400 Mbits/sec/port

A key feature of the machine is the *explicit token storage* mechanism that replaces the associative waiting-matching section with conventional single-port memory. The token storage can also be imperatively manipulated like standard memory while avoiding any data constraints. This feature should provide the needed support for coding resource managers and I/O drivers, as well as opening the architecture to the "multi-threaded" parallel processing paradigms.

2. Translating the TTDA into Practical Hardware

By restricting the operating principles to TTDA-style dataflow, we leverage our existing investment in software tools and analysis. We are being faithful to TTDA if the compilation strategy is the same, the various schemata are closely related, and the internal structure of the machines is roughly equivalent. Meeting the compilation constraint means that the semantics and manipulation of tags are the same. The internal structure of the TTDA can be maintained by providing the same abstract pipeline. Thus, the translation of the TTDA into a practical machine requires developing of a consistent set of *representations* for the values manipulated by instructions and *mechanisms* for the realization of pipeline stages.

2.1. Preserving TTDA Tag Semantics

TTDA tag semantics are a bounded variety of U-interpreter tag semantics. The TTDA tag is essentially a triple of context (U), iteration (I), and statement (S). The GET-CONTEXT operation guarantees a unique context identifier. The iteration field is really an extension to context, providing a structure that can be interpreted by the compiler. One view is that GET-CONTEXT returns a set of unique names parameterized by iteration. This allows code blocks, primarily loops, to generate unique names locally. Ignoring loop constants for a moment, we argue that iteration is an efficiency optimization that is unnecessary if GET-CONTEXT is inexpensive.[4]The statement field specifies the instruction to be executed, relative to the code block associated with the context.

The relationship between a tag and a particular processing element is intentionally obscured from the compiler. The maintenance of this mapping is the responsibility of the run-time system. In general, all three fields, U, I, and S, are used to determine the PE number. The mapping of all statements for a given context and iteration onto a single processor[5] has been explored with good results. However, to require all iterations of a loop to be on the same processor appears to be overly restrictive. Even so, it is probably premature to design-out subdomain sizes greater than one. Accordingly, we propose a tag/PE number scheme that can be made sensitive to statement number.

2.2. The TTDA Abstract Pipeline

We intend to preserve the general pipeline flow of the TTDA. This pipeline exploits a non-blocking execution of enabled activities, potentially yielding very high performance designs. This pipeline is abstract to the extent that the mechanisms employed at each stage are unspecified. Moreover, the stages are not particularly well balanced. As shown in figure 2-1, incoming tokens enter the waiting-matching section. The result will either be a match, in which case two tokens are emitted, or a non-match, meaning that no tokens are emitted and the token waits for its partner. Unary operations bypass this section. In the instruction-fetch stage, the tag-specified statement is fetched from the instruction store. ALU operations may be performed simultaneously with destination-list processing. New tokens are assembled in the form-token stage. Finally, a large token queue provides buffering for excess parallelism.

[4]We argue further that consuming a name for a context on demand is a more efficient way of managing a physical namespace. If the names were virtual (*i.e.*, physical resources are not necessarily bound to the name), then we would reverse our position here.

[5]That is, subdomain size is one.

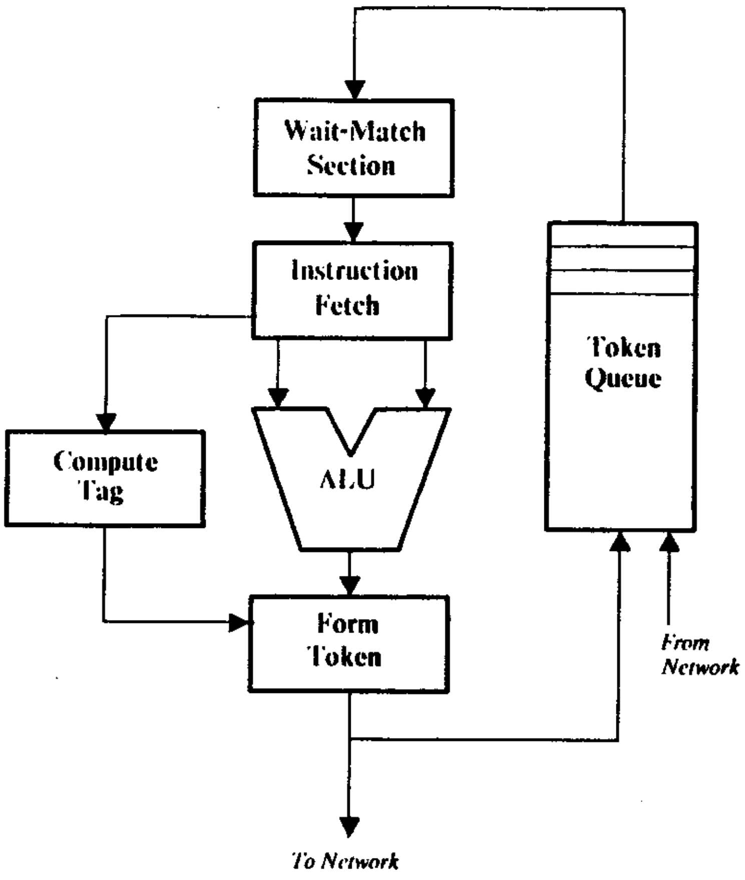


Figure 2-1: The TTDA Abstract Pipeline

2.3. First Order Implementation Issues

The translation of the abstract pipeline into a practical design raises a host of design issues. The most difficult issues to resolve are ones concerning resource limits. Taking very conservative upper bounds at this time might make the design unwieldy. However, "blowing out" a resource might seriously undermine the utility of the machine. We attempt wherever possible to parameterize particular resource limits at run-time as a subdivision of some larger resource pool. For instance, the relative amounts of instruction and token storage have not been carefully studied. By having one memory for code and tokens (with appropriate caches), we can translate this into a single resource limit, the amount of primary storage per processor, and we make it as large as feasible (two to eight megawords).

The issues which have a first-order effect on the design are as follows:

1. **Pipeline Balancing.** Are the stages composed of roughly the same levels of logic? Are the flow rates through the different stages consistent?

2. **Resource Limits.** There are many finite resources to exhaust. The following parameters

need binding:

- Waiting-matching store size.
- Token queue total size.
- Maximum number of contexts.
- Maximum number of code blocks.
- Total size of code blocks.
- Maximum number of structures.
- Total size of structures.
- Total size of deferred read queues.

3. **Representations.** Should data values be of variable or fixed-size? Can instructions be made fixed size with a fixed number of destinations?

4. **Input/Output.** What kind of I/O is best? Direct I-structure access through channels? Input/Output instructions in the processor? Memory (token or I-structure) mapped?

2.4. The Waiting-Matching Problem

A clear virtue of the dataflow scheduling mechanism is the non-blocking pipeline once an activity has been enabled. However, a fundamental problem arises when trying to implement the waiting-matching section. Not only must the waiting-matching have a relatively large token capacity, but to balance the pipeline properly, it must also have a throughput up to *twice* that of the ALU. An efficient implementation of the waiting-matching section is central to the goal of translating the TTDA into practical hardware.

If a graph consists mainly of dyadic operators, then most instructions require two inputs and, on average, produce two destinations. The destinations of an instruction are processed independently by the wait-match section. So the ALU can consume and produce two values in a single time step, but the wait-match section takes two time steps to consume two values. Similarly, the wait-match section, on average, can produce only one value per time step. In this scenario the ALU is *at best* fifty percent utilized. In practice we see about seventy percent utilization due to the large number of identities used in termination trees.[6]

There are three approaches to solving this imbalance. The first, and easiest, is simply to accept it

[6] It might be (convincingly) argued that identities shouldn't be counted as ALU operations since they are strictly overhead. The fact is they consume ALU cycles so for the purpose of pipe balancing a lot of identities really do help. Better compiler technology ought to reduce the number of identities, though, so we head back to fifty percent.

and assign it to copying overhead inherent in data-driven computation. Second, make a wait-match section that runs twice as fast as the ALU. If you take the technological scalability goal seriously, this is really achieved by *slowing down* the ALU to half of the waiting-matching speed. Finally, we can look for wait-match structures that have internal parallelism. In essence, multiple or interleaved wait-match sections.

This problem might seem a bit contrived, but the reality is many times *worse*. Although fully associative wait-match sections were proposed first, hashing schemes are actually employed in almost all designs. Even the very best hashing function requires the following steps for a wait/extract operation:

1. **Hash and Compare.** Hash the incoming tag, read the hashed-to location.

2. **Insert.** If the location is not occupied, then insert (write)

3. **Extract.** If the location is occupied then, extract (write empty)

It appears the even this ideal hashing scheme takes two cycles. As soon as collision logic is employed (this complicates the extract), only the very best of implementations can boast three-cycle performance. In this case a pipelined ALU with a single cycle throughput will be utilized less than twenty percent of the time.

3. The Explicit Token Storage Model

A novel approach to the wait-match problem is to treat the wait-match section as ordinary random access memory where each word has a presence bit and which is allocated and deallocated with procedure activation and deactivation, respectively. This *explicitly* managed token storage can be efficiently implemented as a multi-stage, single-cycle throughput pipeline that employs conventional memory technology.

This approach was jointly developed by David Culler and the author, and largely inspired by the VIM dataflow model of Dennis, Stoy, and Guharoy.[7] As such, the mechanism is related to the static dataflow architecture of Dennis. The semantics, however, are firmly entrenched in the TTDA and thus the U-interpreter of Arvind. The synthesis presented here relates the fundamentally more expressive tagged-token model to the implementation efficiencies of the static paradigm. To the extent that our solution is also completely pipelined with only single-port access to the token storage, we believe it to be a significant advance in dataflow machine engineering.

[7]"VIM: An Experimental Multi-User System Supporting Functional Programming." *Proceedings of the International Workshop On High-Level Computer Architecture 84*, pp. 1.1-1.9

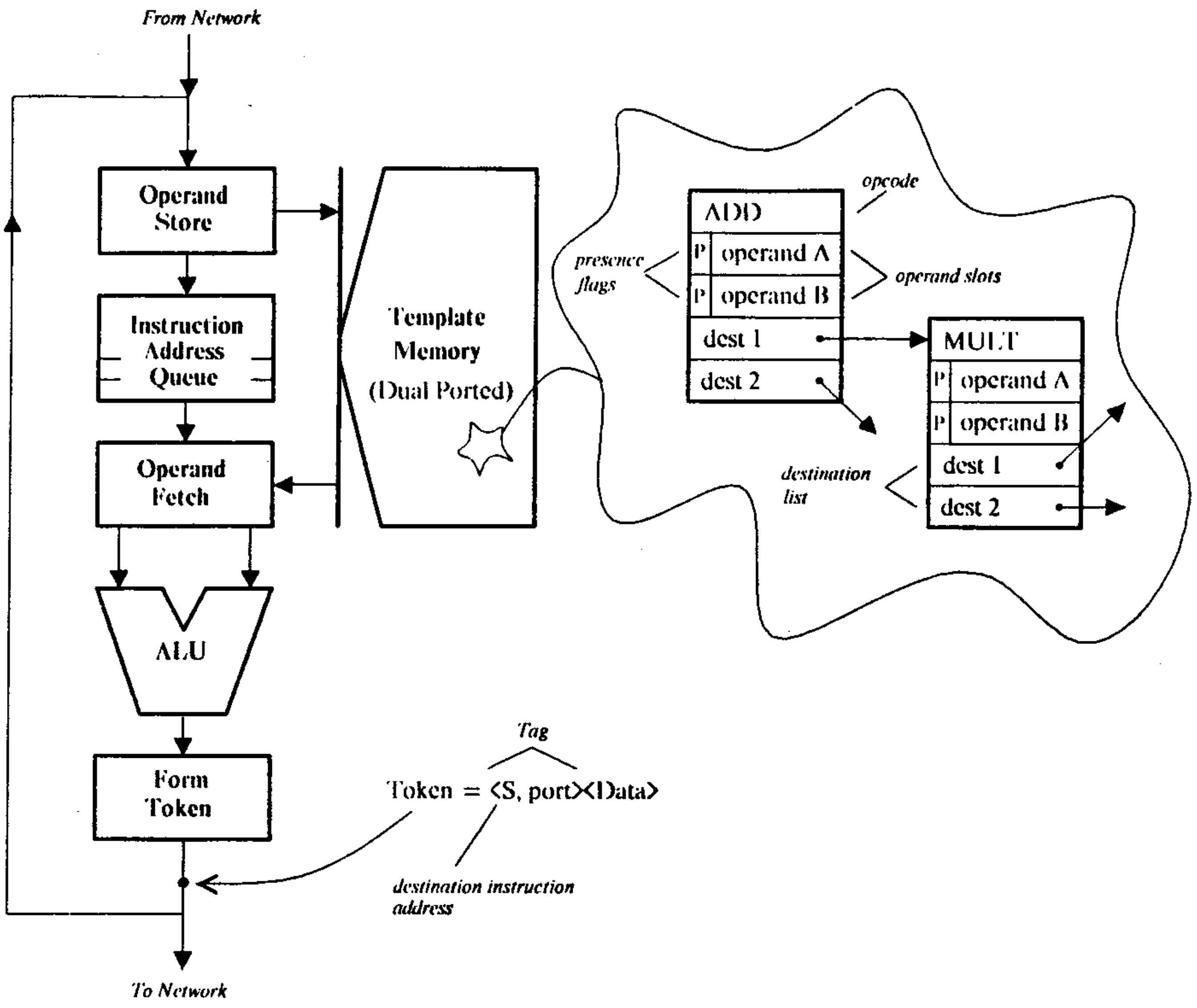


Figure 3-1: A Dennis-Like Static Dataflow Machine

3.1. The Static Dataflow Machine.

Dennis' static machine is distinguished from the tagged-token architecture by its restriction to one-token-per-arc. This permits token storage allocation prior to execution, since the number of arcs is fixed for any graph. The storage required by a token traversing a given arc can be *statically* allocated in a conventional memory. Each such location is called an *operand slot*, and each slot has a *presence flag* to indicate whether or not a value has been stored. The slots are conveniently related to the instruction template as shown in figure 3-1. Thus, when a token is stored, it is straightforward to determine if all the other required operands for the associated instruction are present. If so, the address of the instruction template is inserted into an instruction address queue. The execution cycle of an enabled instruction involves fetching the operands from the template (token) store, performing the indicated operation, and distributing the results as specified by the instruction's destination list. For the purposes of this discussion, we assume at most two operands per instruction and ignore acknowledgments.

A tag in this machine is simply represented as `<S, port>`, where `S` is a physical address of a template in the template store[8] and `port` identifies the operand number within the destination template. This machine is relatively efficient and employs conventional technology. Superficially, it seems to pipeline well, but only assuming that the template storage is truly a dual port memory. This is a troubling assumption because the template memory is apt to be large, and large dual port memories are relatively slow and expensive. If this memory is to be implemented as a multiplexed single port memory, then the pipeline can only run half as fast as the memory access time, and a purely dyadic instruction rate is four times slower than the memory access time. This is because each dyadic instruction requires four memory accesses; two to store the operands and two to fetch them.

The static architecture is fundamentally less expressive than the TTDA model in that the one-token-per-arc restriction prohibits the recursive application of a function and precludes dataflow graphs, which are first-class objects. One solution is to *copy* the instruction templates associated with a function for each application; every time a function *f* is evaluated, the code for *f* is copied to a region of memory and the particular arguments are inserted into the new graph. Such a machine would no longer be strictly static as the solution implies a run-time manager of the template storage. Note that the destination lists can no longer refer to absolute instruction addresses, but must instead be *relative* to the particular instantiation of *f*. One tag format would be `<C,S, port>`, where `C` points to the base of the new *f*, and `S` is a statement number that selects a template relative to `C`. The `port` selects one of two operand slots. This contextual information is analogous to the TTDA tag. Working out the details of argument/result linkage, token-store (tag namespace) management, and unfolding control are the same challenges facing a tagged-token machine. The obvious deficiency of this approach is the excessive overhead implied in the need to copy the graph every invocation.

3.2. Separating Templates and Slots

One easy way to mitigate the copying overhead is to separate the static part of an instruction template (the opcode and destination list) from the operand slots. See figure 3-2. Here an invocation of *f* involves the allocation of a set of slots for the corresponding instruction templates. The tag contains four items, `<C,R,S, port>`, where `C` is a pointer to the instruction templates of *f* (*sans* operand slots), `R` points to the base of the operand slots for an instance of *f*, and `S` is a statement number that selects an instruction relative to `C` and a pair of operand slots relative to `R`. So the address of the instruction is given by `C+S`, and the address of the associated slot is given by `R+S`.

A token is processed as follows:

[8] The processing element number is implicitly embedded in `S`. That is, `S` is a global address.

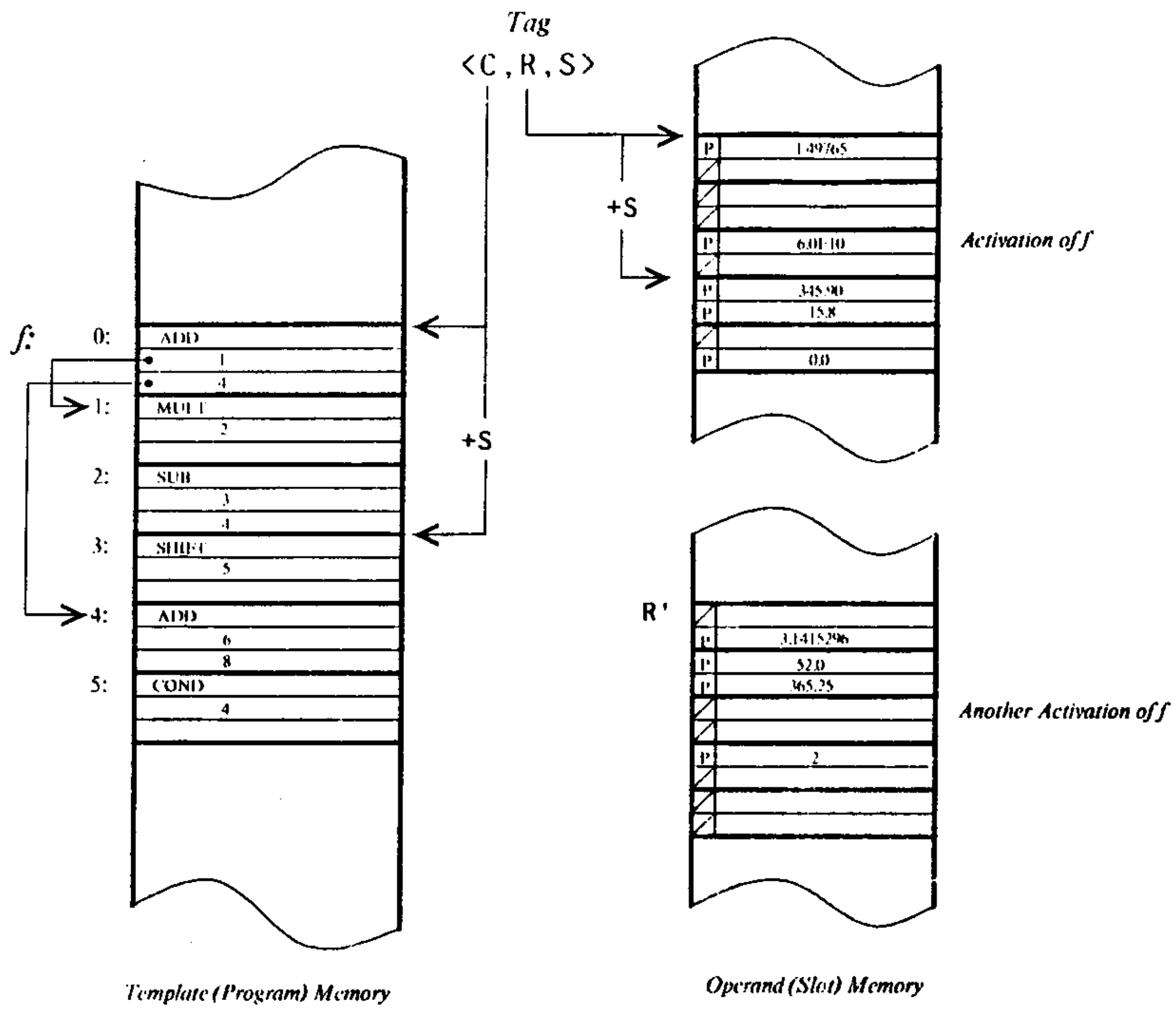


Figure 3-2: Separating Templates from Operand Slots

1. **Store Operand.** Add **R** and **S** to compute the effective address of a pair of operand slots. Write the data part of the token into the appropriate slot as indicated by **port.**

2. **Queue Tag** If both operands are present, then queue the instruction/data addresses, **<C+S,R+S>.**

3. **Fetch Operands.** Dequeue the instruction and operand addresses, fetch the instruction, destination list, and the two operands.

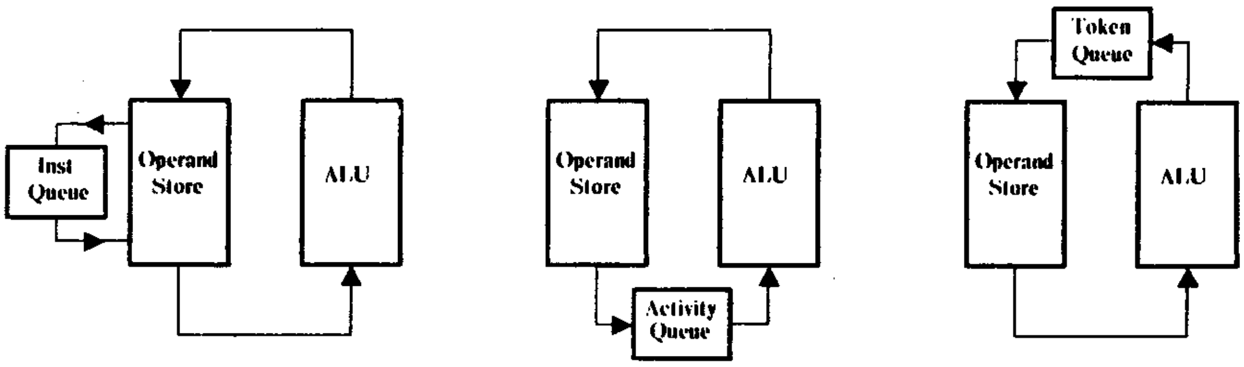
4. **Execute.** Perform the indicated operation on the operands and produce a result value.

5. **Process Destinations.** Construct result tokens by substituting new **S** and **port** fields as indicated by the destination list, substituting the original code-base, **C**, and operand-base, **R.**

This approach does eliminate the copying overhead of the templates, but still requires four slot operations per instruction. In addition, the operand memory is sparsely populated with tokens. Most slots are empty. Perhaps there are more efficient uses of this physical resource.

3.3. Queue Tokens, Not Addresses

Any dataflow processor pipeline is cyclic and, because operators may have more outputs than inputs, some form of queue is required to prevent the processor from deadlocking. The queue accommodates the "excess parallelism" beyond the number of simultaneous activities that keeps the pipeline completely busy. In the static machine above, this queue was in the form of an enabled instruction FIFO; the entries in the buffer keep track of those templates that have enough operands to fire. As shown in figure 3-3, there are two other places to insert a queue in the pipeline.



A. Instruction/Operand Addresses B. Enabled Instruction and Operands C. Unmatched Tokens

Figure 3-3: Avoiding Deadlock with Instruction, Activity, and Token Queues

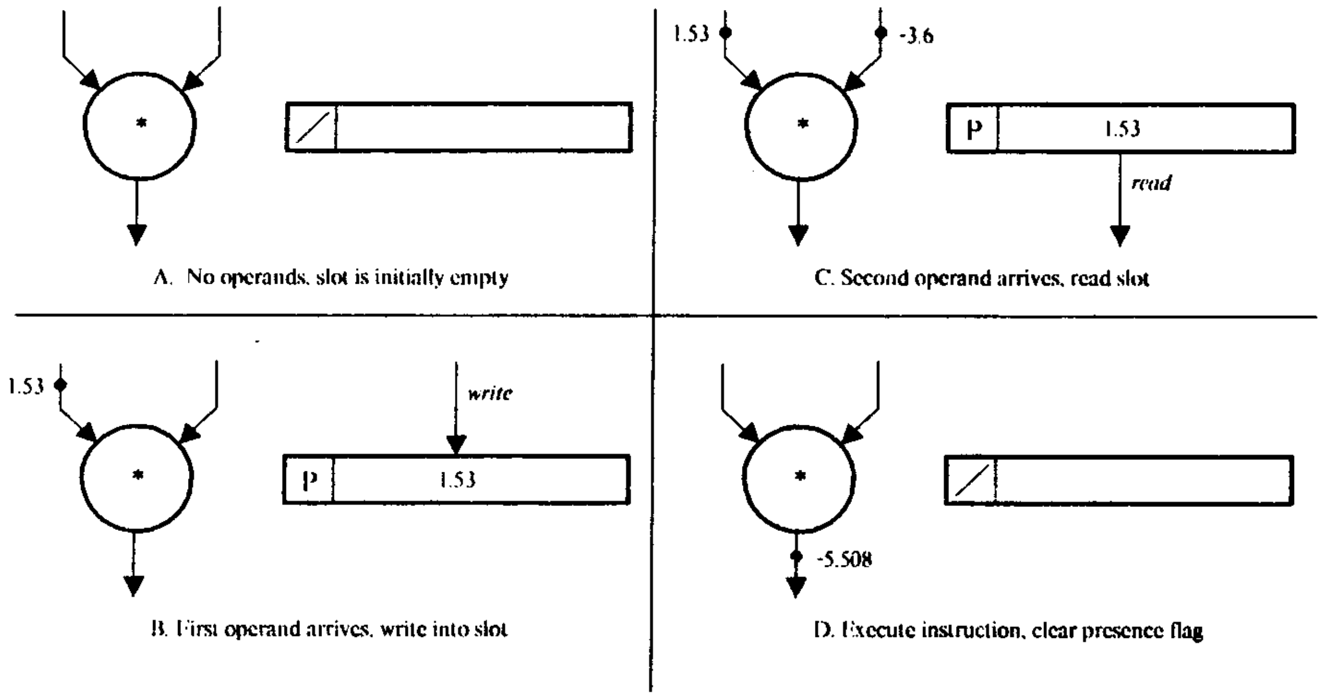
Suppose that when an operand is written it is discovered that the associated instruction becomes enabled. Instead of queueing the address and then later fetching the operands, the operands are immediately extracted. In this case there is no need to write the second operand just to fetch it again. Rather, the first operand is written into the token store and when the second one arrives, the first operand is read. This cuts the number of operand accesses in half; an operand is either read *or* written each cycle.

Under this model the excess parallelism can be absorbed immediately after the operand store in an *enabled activity queue*, a queue that holds the operands, instruction, and tag. Alternatively, a queue can be placed immediately after the ALU in a *token queue*, a buffer that holds result tokens consisting of tags and data values. On balance, the storage requirements of both queues are roughly equivalent. The activity queue is about twice as wide as the token queue but, assuming two destinations per instruction, the token queue has twice as many entries.

The decision of which type of queue to use, activity or token, is really an implementation issue. While we prefer the token queue, the important contribution of either approach is to *convert the operand store into a single-port memory that is accessed as a read or write at only one pipeline stage.*

3.4. Eliminating Half of the Operand Slots

Another advantage of the single-port operand store is a simple optimization that can eliminate half of the operand slots. At any time, we have to store at most one operand of a two-input instruction. When the first token arrives we store its value into the slot computed from its tag and enable the associated presence bit. When the second operand arrives the presence bit is inspected and is found to be set. Now the slot is read and the presence bit is cleared. Figure 3-4 illustrates the firing sequence of a two-input operator using a single slot.



*
A. No operands, slot is initially empty
1.53
-3.6
P
1.53
read
C. Second operand arrives, read slot
write
B. First operand arrives, write into slot
-5.508
D. Execute instruction, clear presence flag

Figure 3-4: Firing Sequence of Two-Input Operator With Single Slot Rendezvous Point

3.5. Instruction-Specified Slot Addresses

The slot represents a rendezvous point for the instruction rather than storage for an arc. Under the model presented so far, there is a one-to-one correspondence between the instance of an instruction and a slot. That is, the statement offset `S` is used to index the code as well as the operand store. We can extend this idea by permitting the destination instruction to specify the slot to use rather than deriving it from `S`. This actually simplifies the tag while permitting a more general and efficient use of operand storage.

Consider the tag `<S,port,R>` where `S` represents a global instruction address, and `R` represents a global slot *base* pointer. The destination instruction provides `r`, a relative adjustment to `R`, in order to compute the slot address `R+r`. Because the slot firing rule is self-cleaning, a particular slot can be re-used within a given activation. By providing other firing rules, as we shall see shortly, a mechanism is provided for the efficient sharing of constants.

One obvious implication of this model is that instruction fetch must occur *before* the waiting-matching operation. This new pipeline and memory model is shown in figure 3-5. The destination instructions are computed by making adjustments to **S** as specified by the destination list. The new tag is constructed from the old by using the same **R**, incrementing **S** by the instruction-specified **s**, and substituting the instruction-specified **port**.

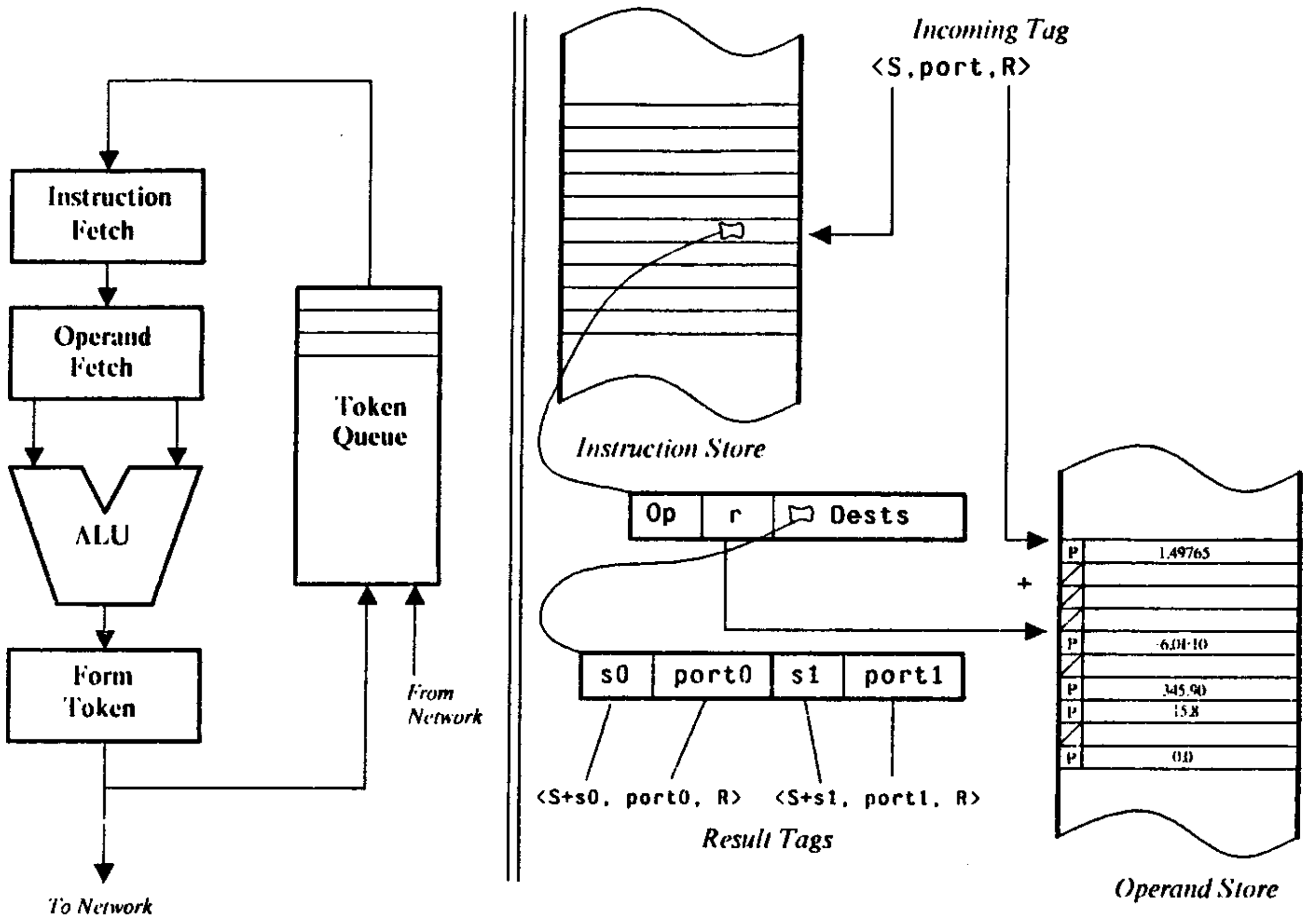


Figure 3-5: Explicit Token Store Pipeline Overview and Memory Model

3.6. Instruction-Specified Slot Operation

Prefetching the instruction allows more control over the operand fetch (waiting-matching section). The destination instruction could specify an operation different from the standard self-cleaning firing rule. For example, leaving the presence bit set after a match creates a "sticky" or constant value. In general, the instruction specifies a *waiting-matching opcode* such as NORMAL, READ, WRITE, NOP (unary), CONSTANT, *etc.* An important aspect of any waiting-matching opcode is whether the ALU operation and destination processing is performed conditionally. In the NORMAL firing rule, the ALU operation is conditioned upon the presence of both operands. But the READ, WRITE, and NOP cases *unconditionally* execute the ALU operation. This gives rise to a different view of a dataflow machine; instructions are executed unless *inhibited* by a synchronization constraint.

We can further generalize the operand store by associating more than one bit of state for each location, permitting more states than empty/present. In general, the waiting-matching opcode specifies a state-transition function

```
new-state, alu-inhibit, read-write := WAIT-MATCH-OP(state, port),
```

where `new-state` is the new state to be associated with the slot, `alu-inhibit` is a predicate that can inhibit further instruction execution, and `read-write` specifies an operation on the data part of a slot: read, write, or exchange. Notice that `port` is used as an input to the state machine. The proposed machine has two bits of state associated with each slot.

The explicit specification of the slot operation permits highly imperative control of machine resources, if desired. One can view `R` as a *frame pointer* to a set of synchronizing registers. The register number relative to the frame pointer is specified by `r`. The data value on a token can be selectively written into a register, or the contents of the register may be read and combined with the incoming data value to produce a new result data value. Thus, the data value carried on a token is analogous to an accumulator, the `S` field an instruction pointer, and the `R` field a frame pointer. An instruction with two destinations creates a *fork*, the `R` and accumulator are copied and a new `S` is supplied. A dyadic operator that specifies a NORMAL wait-match opcode is effectively a *join*.

4. Compiling Dataflow Programs for an Explicit Token Store

The explicit token store provides a TTDA-equivalent programming model for acyclic code blocks. A token is composed of a tag and a datum:

```
Token  =  <S,port,R><Datum>
```

The `R` component of the tag `<S,port,R>` is a pointer into the machine's operand memory, while the `S` part is a pointer into the instruction memory. Thus, a tag is also a pointer with two components: a data pointer and an instruction pointer. The normal tag construction rules only involve the manipulation of `S` and `port`. `R` is simply passed through to the result tags. `R` is thus the *context* of the particular invocation of the block referenced by `S`, equivalent to the U-interpreter context, `U`.

The only difference in code generation for acyclic blocks is the assignment of the `r` field for each instruction. A simple approach is to assign a unique `r` in a one-to-one correspondence with the instructions in the code block.

4.1. The SEND instruction.

Each invocation of some function *f* requires (1) the allocation of a new context R_f, (2) the transfer of arguments to the function, (3) the gathering of results from the function, (4) the termination detection of the function, and (5) the reclamation of the context.

Suppose, for now, that step (1) has been accomplished. That is, a program has somehow obtained a fresh area of token memory as pointed to by R_f. A tag can itself be a datum (of type tag) manipulated by a program, so a tag is constructed of the form $\langle S_f$,port,$R_f\rangle$, where S_f is the first instruction in the function we want to invoke. How are we to supply the arguments, say *x* and *y*, to this instance of *f*?

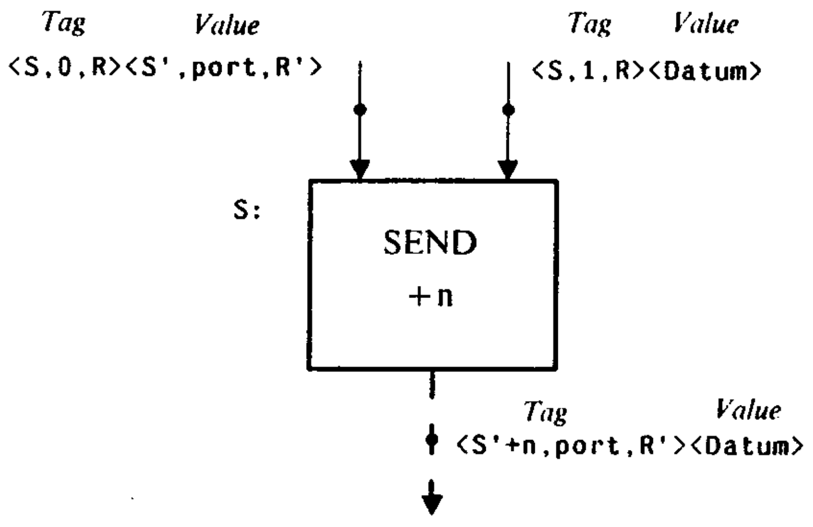


Figure 4-1: The SEND Instruction

The basic requirement is the ability to take a value from the current context and *send* it to another context. If the target context is represented by a value of type tag, all that is required is the construction of a token whose tag is this value and whose value is the argument we wish to send. As shown in figure 4-1, the SEND instruction is a dyadic operator whose inputs are a tag type and an arbitrary value and whose output is a token constructed from the supplied tag and value. The SEND instruction is parameterized by n, an optional offset to add to the S part of the tag argument.

In order to send the arguments *x* and *y* to the correct instruction in *f*, a linkage convention is required. Suppose we agree to send the first argument to S_f+1, the second to S_f+2, and so on. Most likely, these instructions are just identities, so port is irrelevant. But how do we get the results back from *f*? Quite simply. We supply *f* our "return address." That is, we send a token to *f* whose value is a tag in *our* context where we want the return values to be sent. If our context is R, then we construct a tag of the form $\langle S_a$, port, R$\rangle$, where S_a is an instruction in our code block. By linkage convention this return tag is sent to S_f+0.

Results are sent back to our context by SEND instructions in *f*. By linkage convention the first result is sent to S_a+1, the second to S_a+2, *etc.* Finally, the termination signal from *f* is sent to S_a+0. This whole operation is shown in figure 4-2.

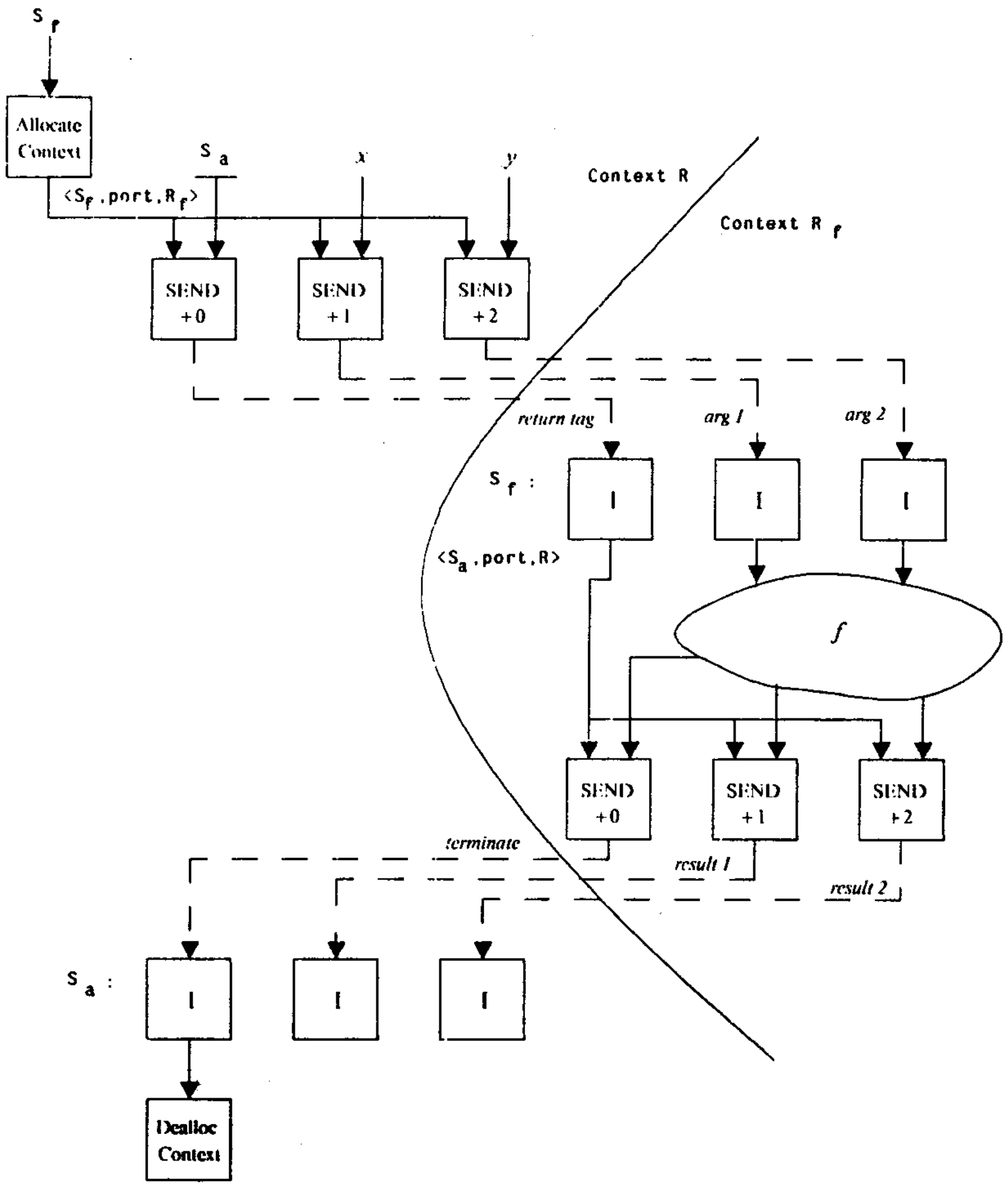
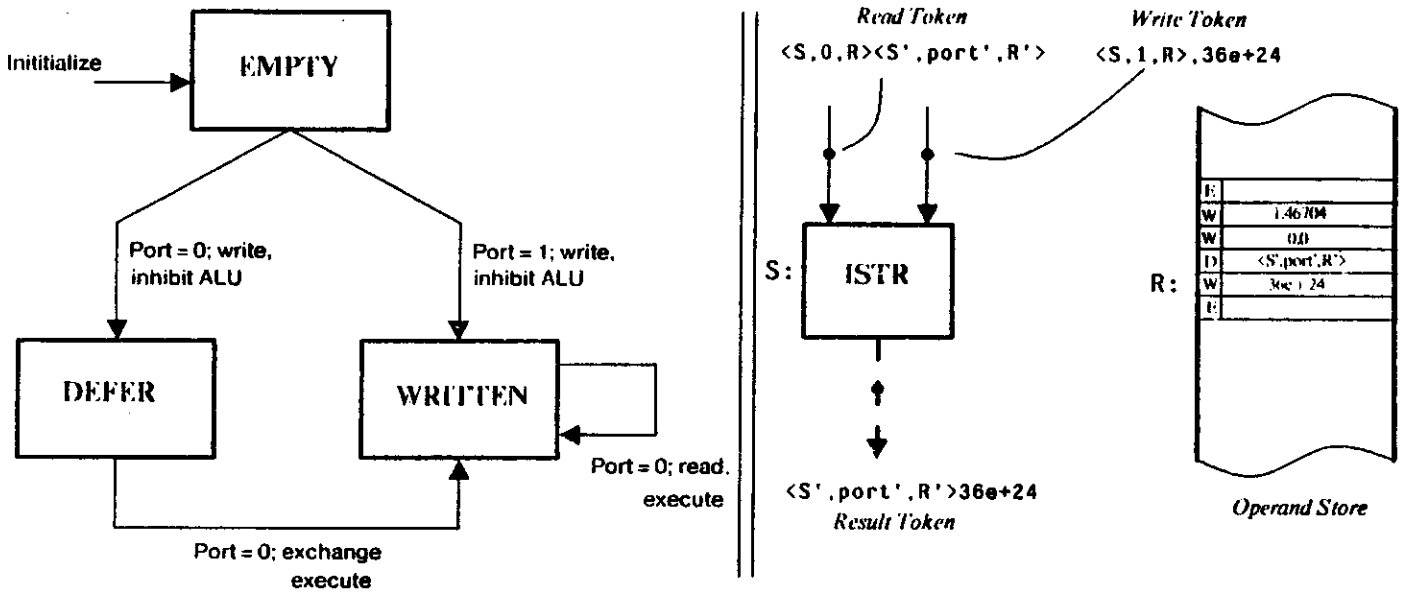


Figure 4-2: Calling an Instance of *f*.

4.2. Implementing I-Structures

In the preceding discussion, we viewed **S** as pointing to the code for a function while **R** pointed to an activation frame. Alternatively, we view **R** as a simple pointer somewhere into memory while **S** provides a limited set of instructions for viewing the the location pointed to by **R**.

For example, suppose that **S** points to an instruction that reads the zero-offset location of **R**. The data part of the token is a tag where the result is to be sent. The instruction unconditionally reads the location and then constructs a result token whose tag is the incoming token's data part and whose data part is the value read. Thus any function could read a location in memory by sending a return address to the location to be read and specifying the **S** that performs the read function. This is a classical, and completely general, split-transaction memory read. Similarly, a function can write any location in memory by sending the data to be written to the desired location and specifying an **S** that performs an unconditional write.



A. Wait-Match State Machine for ISTR

B. The ISTR Instruction and an I-Structure Vector

Figure 4-3: I-Structure Operator Without Deferred Read List

The presence bits associated with each memory location permits the construction of *synchronizing reads*, completely analogous to an I-structure. As shown in figure 4-3 a new operator, ISTR, can be constructed along with a new state machine for the wait-match operation. Writes are sent to **port 1**, read requests are sent to **port 0**. If the write arrives before the read, then subsequent reads find a value already in slot; the value is read (but the presence bit is not reset), a result token is formed and the requested value is sent back. If the read arrives before the write then the return address is written into the slot. When the write arrives the the slot is *exchanged;* the deferred read address is extracted, the write-value is written, and a result-token is formed. The ISTR instruction is simply a SEND instruction with one port designated as constant or "sticky" value.

Deferred read lists can be handled in several ways. One possibility is to provide for the hardware maintenance of deferred read lists in the operand store itself. However, this approach does not pipeline well and antagonizes the RISC philosophy underlying the machine design. We prefer solutions that place the defer burden on the *requesting instruction* rather than the requested location. One such approach requires a convention whereby the requesting context allocates a slot for each potential outstanding deferred read. The ISTR instruction can be augmented with an additional state to cause the queueing of extra read requests through these slots. Very simply, if a location is read where it is discovered that a deferred read already exists, the existing request is extracted and sent to the reserved slot of the current request. The current request is first modified so that it points to the deferred slot (for instance, by incrementing S), and then written into the ISTR slot. When the desired value finally arrives, it is forwarded to the reserved slot, which forwards it along to other readers, as well as copying the value for its own use.

5. Representations

The processor data paths are logically 72 bits, 64 bits of data and 8 bits of type. All tags, floating-point numbers, bit fields, and integers are 64 bits, yielding a single, fixed-size quantity for all operations. Instructions are a fixed 32 bits, with two instructions per word. This greatly simplifies the processor and network design at the expense of space efficiency for certain representations. Figure 5-1 illustrates the relationship between tokens, tags, data and instructions.

5.1. Basic Data Types

All data in the machine are represented with a fixed 72 bit word, 64 bits of value and 8 bits of type as follows:

```
Datum  =   <Type> < Value >
              8      64
```

There are only four basic hardware types:

1. **Number.** 64 bit IEEE double precision floating point. Integers are distinguished only as non-fractional floats.

2. **Bits.** Uninterpreted bit field of 64 bits. Standard operations of shift, mask, and of two's complement addition and subtraction.

3. **Tag.** 25 bits of instruction address and 32 bits of data address point to an **<Instruction, Datum>** pair.

4. **Error.** The value part is meaningless.

Other types like Boolean, Acknowledges, Closure, *etc.* are actually one of the hardware types with the compiler maintaining certain conventions. The remaining six type bits are compiler-definable.

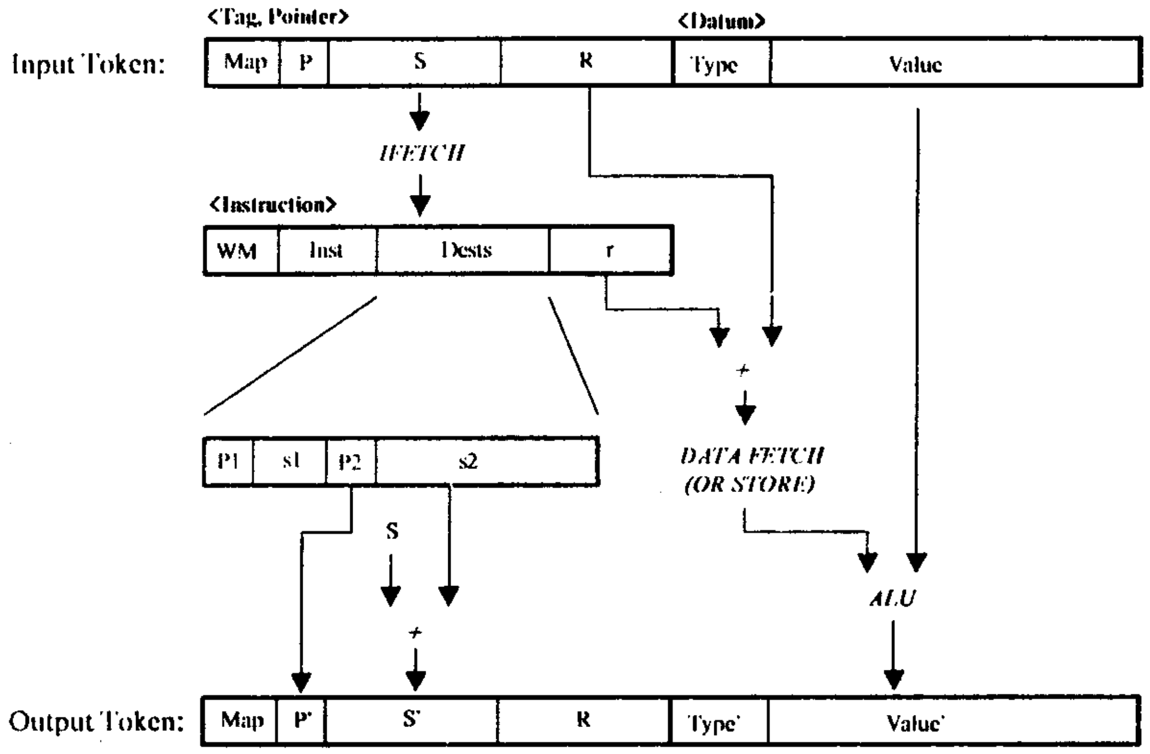


Figure 5-1: Token, Tag, Data, and Instruction Representations

5.2. Using Tags as Pointers

A token is simply a `<Tag, Datum>` pair:

```
        Token  =      <Tag><Datum>
                        64     72
```

A tag has the internal structure:

```
             Tag   =  <x> <Map> <Port> <S> <R>
                       1    5     1    25  32
```

A tag describes a *pair* of values: a Datum offset and an Instruction offset. When thought of as a traditional tag, the instruction part refers to the activity to schedule when the specified data dependencies are met while the datum part refers to the base slot of the context in the waiting-matching section. When thought of as an I-structure pointer, the instruction part refers to an I-structure operation while the datum part describes a linear offset into the structure. The encoding of the different fields are as follows:

1. `<X>`. This bit is presently undefined.

2. `<Map>`. Defines the mapping operation for the `<R>` field:

   ```
        Map  =  <Extent> <Alias>
                    4       1
   ```

   The `<Extent>` field defines $\log_2$ the number of PEs in the subdomain referenced by the data part of the pointer. The `<Alias>` bit indicates that any processor defined within `<Extent>` can receive any token. See the description of the `<R>` field.

3. `<Port>`. The destination port, LEFT or RIGHT.

4. `<S>`. The instruction offset (Statement) number. A physical pointer local to the PE selected by `<R>` and `<Map>`.

5. `<R>`. Describes a global physical data address. All address arithmetic treats this as a single linear address space, but the hardware uses the `<Map>` field to determine the PE number. When no `<Alias>` is indicated:

   ```
        R  =  <PE-hi> <R-local> <PE-low>
              8-Extent   24      Extent
   ```

   When `<Map>` is zero, this degenerates to a subdomain of size one, a purely local pointer:

   ```
        R  =  <PE> <R-local>
               8      24
   ```

   On the other extreme, when `<Map>` is eight, this degenerates to a global address with an interleave factor of 256:

   ```
        R  =  <R-local> <PE>
   ```

```
                         24        8
```

When `<Alias>` is selected, only the computation of the `<PE>` field is affected:

```
R  =  <PE-hi>  <X>     <R-local>,
      8-Extent Extent    24
```

where `<X>` defines a don't care for the least significant `<PE>` bits.

5.3. Instructions

Instructions are fixed size, 32 bits, with one or two destinations:

```
Instruction = <Inst> <r> <Dests>
                9     10   13
```

1. `<Inst>`. Selects one of 512 machine nano-instructions. The nano-instruction table is downloadable, the format being described below.

2. `<r>`. A two's complement adjustment to the tokens `<R>` field. The data address is computed as `<R> + <r>`. This field can be used as a short immediate value when `<WM>` is NOP.

3. `<Dests>`. Describes one or two destinations as relative adjustments to the current instruction number `<S>`. A single destination is encoded as:

   ```
   One-Dest = <0> <Port> < s >,
               1     1     11
   ```

   where `<Port>` is the destination port and `<s>` is a two's complement adjustment to `<S>`. Two destinations are encoded as:

   ```
   Two-Dest = <1> <Port1> < s1 > <Port2> < s2 >
               1     1      4       1       6
   ```

5.4. Nano-Instructions

A Nano-instruction is a fully decoded control word that is selected by the `<inst>` field of the executing instruction. Nano-instructions are stored in a downloadable table for maximum flexibility. While the detailed encoding and size of a nano-instruction is not presently specified, it does have the following overall structure:

```
Nano-Instruction = < WM Control >< ALU Control > < Output Control >
```

The `<WM Control>` describes the wait-match operation by providing a pointer into a table of state machines, which describe the transition from the current presence state of slot to its new state, given the port of the input token and whether the instruction should be executed. There are four basic state machines to be used: SYNC/READ provides the standard self-cleaning firing rule, the instruction is conditionally scheduled upon the arrival of data. WRITE unconditionally writes the incoming value into the slot and always schedules the instruction. LITERAL uses `<S> + <r>` to define a code-relative compile time constant. NOP ignores the wait-match section and unconditionally executes the instruction, in which case the instruction's `<r>` field can be used as a signed immediate value.

The `<ALU Control>` specifies the ALU and FPU opcodes. The `<Output Control>` specifies the construction of output tokens:

```
Output Control = <Cond> <EP1> <FT1> <EP2> <FT2>,
                   3      1     2     1     2
```

where

1. `<Cond>`. Selects the condition code for the emit-predicates, `<EP1>` and `<EP2>`.

<Cond>	Asserted When ;
ALWAYS	Always true
ZERO	ALU output is zero
POSITIVE	ALU output is greater than zero
NEGATIVE	ALU output is less than zero
WM-COND	WM condition code = 1
TYPE-MAP	Condition bit from type map = 1

2. `<EPn>`. Specifies the "emit-predicate" for destination `n`. If equal to one, a token is emitted only when the specified condition code is true. If equal to zero, a token is emitted only when the specified condition code is false.

3. `<FTn>`. Specifies the form-token operation for destination `n`. The ALU outputs two results: The ALU output (or operand A, as controlled by `<M>`) and operand B. These results can be combined with the tags computed by the destination processing to make tokens of the structure `Tag:Datum`

FTn	n = 1	n = 2	
0	Tag1:ALU	Tag2:ALU	; Normal
1	Tag1:B	Tag2:B	; Operand B
2	ALU:Tag1	B:Tag2	; Send Tag
3	ALU:B	B:ALU	; Send Data

Note that `<FT1>` and `<FT2>` are independently specifiable.

6. Proposed Pipeline

The proposed pipeline block diagram is show in figure 6-1. All pipeline stages run at the same rate with a single wait-match interleave. This limits ALU utilization to a maximum of 50 percent for purely dyadic instruction mixes.

Starting from a token entering the top of the pipeline:

1. **Instruction Cache.** The `<S>` field of the incoming tag is treated as local physical instruction address. The 32-bit instructions fetched from the DRAM are cached in this stage. **Shift.** The `<R>` field is barrel-shifted according to the `<Map>` yielding a local physical data base address `<R-Local>`.

2. **Instruction Decode.** The fetched instruction's `<inst>` field is decoded by table-lookup (512 entries) to yield a nanoinstruction. The nanoinstruction specifies control for each of the subsequent pipeline stages. **Effective Address.** The `<R-Local>` is added to the instruction's `<r>` field to yield a physical data address. If the instruction specifies a LITERAL operand then the physical data address is generated by adding `<S>` and `<r>`.

3. **Presence Bits.** The effective physical data address selects two presence bits from high speed memory. The nanoinstruction specifies a state transition table from the current presence state to the next presence state, using the incoming token's `<Port>` bit as input. The state table also specifies the operation to the data slot (read, write, exchange) and whether the current instruction should be aborted (*i.e.*, no partner) after the data cache operation.

4. **Data Cache.** The effective physical data address selects a word from DRAM and the presence bits stage provides the operation on this word. The full 72-bit data words are cached in this stage.

5. **ALU/FPU.** A vanilla 64-bit ALU and Floating-Point Unit. There are few special operations to deal with the structure of tags, but otherwise routine. **Form Destination.** A trivial calculation of two result tags computed by adding the incoming token's `<S>` field to the `<s1>` and `<s2>` specified in the instruction's destination list. View this as a function unit parallel to the ALU that is specialized for simple tag arithmetic.

6. **Form Token.** The zero, one, or two output tokens are formed by concatenating tags with data. Usually the tags are obtained from Form Destination and the data from the ALU/FPU, but most any composition is allowed. The Form Token stage can emit up to two tokens per step. One token can be designated *queued* which causes it to be enqueued onto the token queue. The other token may be specified as *direct*, which is immediately submitted to the instruction cache. The default for single-output instruction is *direct*.

7. **Token Queue.** In a given timestep the token queue either receives a token from the Form Token stage or the Net, or sends a token to the instruction cache, or is idle. Normally, the token queue is enqueueing during two-destination instructions, is idle during single-destination instructions, and is dequeuing to fill pipeline gaps (that is when an instruction was aborted during wait-match).

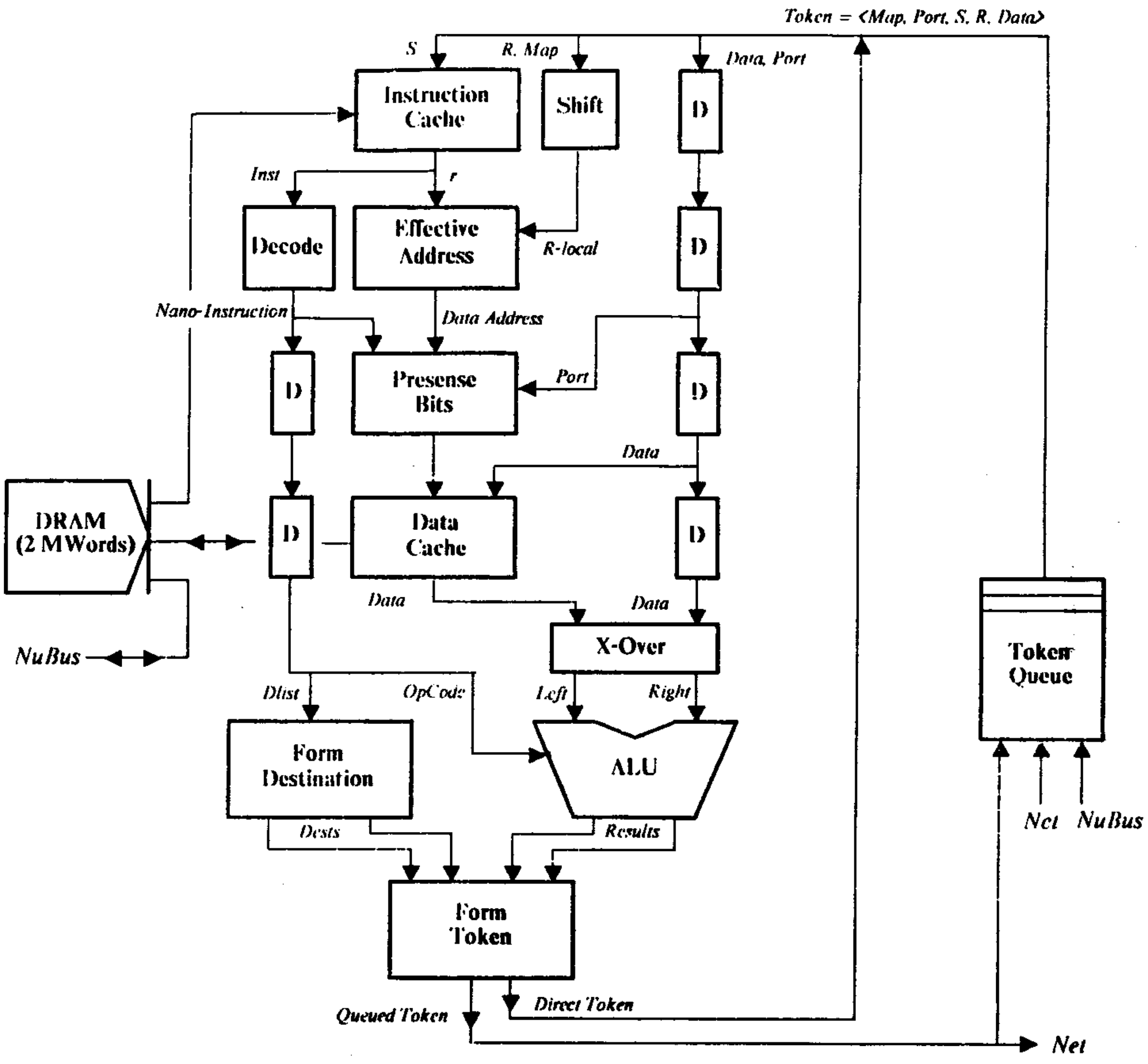


Figure 6-1: Proposed Pipeline Block Diagram