LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Design and Implementation of MINT: A Monsoon Dataflow Simulator

Computation Structures Group Memo 297
June 1989

Andrew Shaw

Submitted to the Department of Electrical Engineering and Computer Science on January 27, 1988 in partial fulfillment of the requirements for the degree of Bachelor of Science.
This report describes research done at the Laboratory of 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 Office of Naval Research contract N00014-84-K-0099.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Design and Implementation of MINT: a Monsoon Dataflow Simulator

by

Andrew Shaw

Submitted to the Department of Electrical Engineering and Computer Science
on May 12, 1988
in partial fulfillment of the requirements for the degree of
Bachelor of Science in Computer Science and Engineering

Abstract

Dataflow architectures can exploit the full parallelism of many algorithms by means of fine grained synchronization. Each instruction in a dataflow graph may be considered a different task which can be executed on its own processor. The principle criticism of dataflow architectures has been the high overhead associated with the distribution and synchronization of the instructions. Most notably, in previous dataflow architectures, there was an apparent need to use either a fully associative memory, or some sort of hashing scheme in order to match operands in an instruction. The Explicit Token Store (ETS) architecture overcomes the need for this bottleneck.

MINT (Monsoon Interpreter) is a bit-level simulator for Monsoon, which is the first hardware implementation of the ETS architecture. MINT is easily extendable to future implementations of ETS. Any state accessible in Monsoon can be accessed in MINT, except the internal state of the pipeline stages. All timing for MINT is identical to Monsoon, including pipeline delays. One can single step through either MINT or Monsoon without telling the difference between the two.

MINT can do several things that Monsoon cannot do. MINT is capable of more detailed statistics gathering than Monsoon. MINT can implement pseudo-instructions which execute several Monsoon instructions; for example, system calls may be implemented as single instructions in MINT in order to minimize the effect of a specific operating system on the statistics gathered for studies of algorithms on dataflow architectures.

Eventually, MINT will be used to help debug Monsoon hardware, prototype new Monsoon opcodes, and it will be shipped to parties interested in performing dataflow experiments.

Thesis Supervisor: Gregory Papadopoulos

Title: Project Manager

Acknowledgments

Ken Traub taught me *almost* everything I know about computers. I have learned more from Ken than I have in any class I've taken at the Institute.

Steve Heller got me my first UROP in the Dataflow Group. Through our many discussions about life and computer science, I have matured as a person and as a scientist.

Jonathon Young helped me write the first prototype of MINT, and answered a lot of extremely dumb questions. Jonathon was always the first person I bugged when I had any questions about Monsoon or MINT.

Victor Zue has been a major influence through my MIT career. Through his initial confidence in me during my freshman year, I have gained the experience necessary to be in the position I am now. Victor has been directly or indirectly responsible for every job I've had since my very first UROP under him.

Professor Arvind and Greg Papadopoulos have been gracious enough to allow me to work under them as an undergraduate. I look forward to our continuing relationship in the Dataflow Group; I hope that I will live up to the expectations they have in accepting me as a graduate student.

The Quint has been very understanding in putting up with a lot of dataflow mumbo-jumbo from me and Derek. Thanks to Derek for our many discussions about dataflow, and thanks for being my roommate for these four years.

Thank you especially Mom, Dad, and Nini, for all of your support and encouragement.

Contents

List of Figures

Chapter 1

The Explicit Token Store Dataflow Architecture

In this chapter, we describe the Explicit Token Store (ETS) architecture, especially in comparison to previous dataflow architectures. It is not meant as an introduction to dataflow and a certain amount of knowledge of dataflow architectures is assumed. In particular, a fairly thorough knowedge of the the MIT Tagged-Token Dataflow Architecture (the basis of the work on ETS) is assumed. For a more thorough introduction to dataflow, the following papers are recommended: [2], [3], [1].

1.1 What is Monsoon?

Monsoon is an implementation of Explicit Token Store, a new dataflow architecture developed by Papadopoulos [10]. Monsoon overcomes many of the implementation barriers of previous dataflow architectures. Monsoon's principle advantage is its ability to match operands in a fast, constant amount of time. In previous dataflow architectures, tokens which did not have a partner were put into a "waiting matching area". Every token would have to check the waiting matching area to see if its partner was there. In hardware implementations, matching was either accomplished with large associative memories (which is prohibitively expensive and slow) or else with some sort of hashing schemes, which did not lend itself well to a high performance, pipelined implementation [8] [7]. In ETS, each token places itself into a predetermined offset of an activation frame, where an activation frame is analogous to a stack frame in conventional Von-Neumann machines.

The offset of each token within the frame is determined at compile time. In this manner, waiting matching storage can be implemented with conventional, tagged memory, and waiting matching can be done in a short, constant time, which allows the architecture to be effectively pipelined.

1.2 Overview of Monsoon Architecture

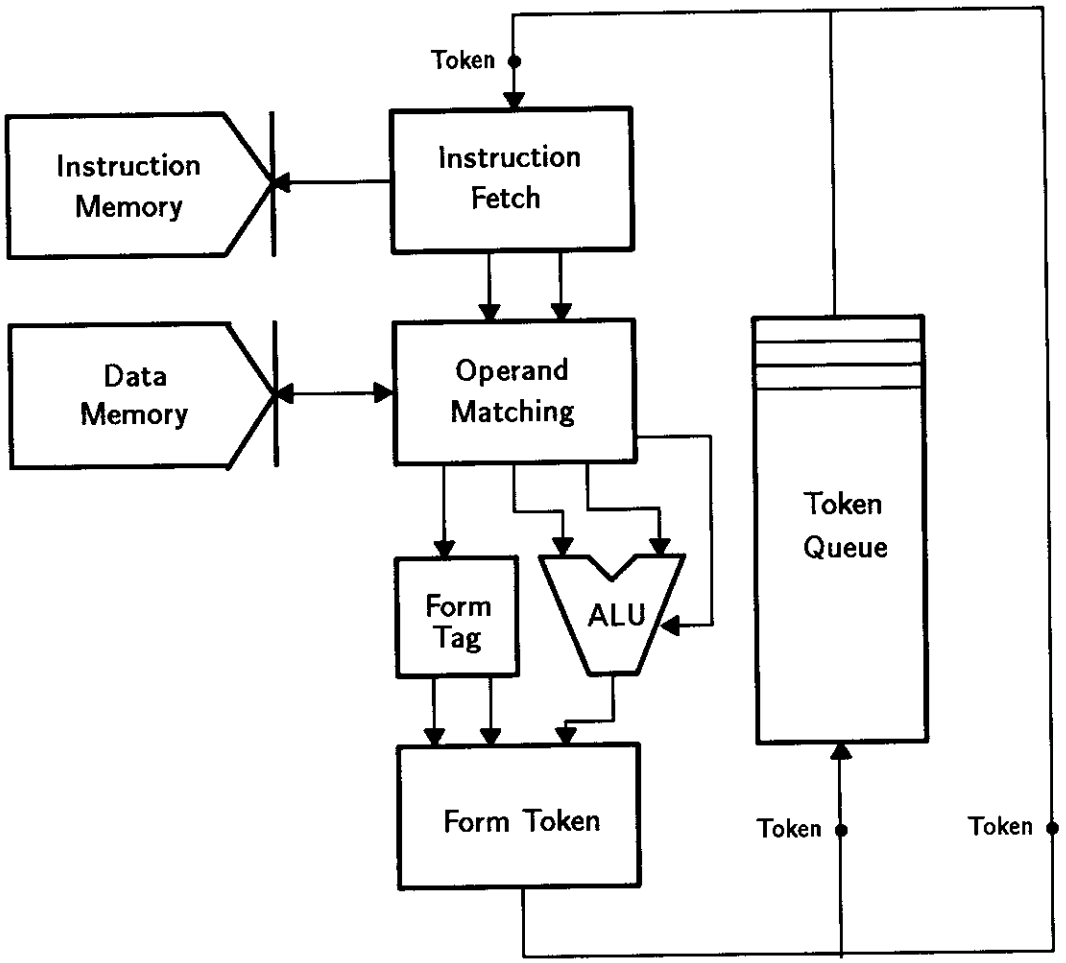

Figure 1.1: The Monsoon Architecture

In Monsoon, data memory and instruction memory are completely separate. Instruction memory is 32 bits wide and data memory is 64 bits of data, 8 bits of tag and 2 bits of presence. The data memory is currently used for I-structure (array) memory as well

as for activation frames and constant storage. Monsoon has two token queues, both of which are LIFO. As stated before, dataflow execution does not depend upon the order in which the instructions are executed, and the queueing policy (LIFO, FIFO, random, etc.) does not affect the determinacy of the program.[1] If the queues were FIFO, the results would be the same, although the amount of time and space required to execute a program might be different. LIFO queues have the attractive feature that they preserve spatial locality; tokens which are processed at about the same time are likely to match and fire, therefore requiring less total memory to execute a given program.

The two queues have slots with 64 bits of tag, 8 bits of tag-type, 64 bits of data and 8 bits of data-type. There are *two* token queues so that an infinite processor simulation can be done in the hardware. One queue can empty out and fill up the other queue; the number of instructions executed until a queue empties is equal to the maximum number of instructions that can be executed in parallel in a dataflow machine with an infinite amount of processors. Once the queue is empty, the roles of the queues are reversed, and this continues until both queues are empty. At that time, the program terminates. The number of times the queues swap is called the "critical path" because it is the minimum number of time-steps in which the particular program can be executed given the that program's inherent data dependencies.

Execution on Monsoon occurs in the following pipelined steps:

- Dequeue Token
- Instruction Fetch
- Operand Match
- ALU and Create Tag or Bubble
- Enqueue Token

Note that the Instruction Fetch stage occurs before the Operand Match stage. This is in the reverse order of the TTDA architecture because certain information from the instruction is necessary to determine how to match operands.

[1] However, certain resource management calls require that a certain tokens be executed before other tokens. For these cases, which will be discussed later, tokens can be inserted directly into the pipeline, overriding the priority of other tokens.

1.2.1 Dequeue Token

Execution begins with dequeueing a token. If the processor is idle, a few "starter" tokens are dropped into the pipeline to start it up. These starter tokens have information as to which function is being called, and the arguments to that function. When the processor is busy, the Enqueue stage of the pipeline usually produces a token which is directly sent to this stage. This is indicated in Figure 1.1 by the direct route from the Form Token stage to the Instruction Fetch stage. Sometimes, tokens arrive from the network to enter the pipeline. If no tokens arrive from the network, and no tokens are directly enqueued from the pipeline, then this stage will unstack a token from the current token queue. If there are no tokens in the current token queue, then the pipeline is bubbled.

Each token consists of four parts: the tag-type, tag-value, data-type and data-value. Currently, the type fields (tag-type and data-type) are not being used, but eventually, the data-type slot may be used for several purposes, including type-checking, garbage collection, and statistics gathering. The data-value contains the data that the token is carrying; this data may be a floating point number, an integer, a bit-array, or a tag (being used as data). Usually, it contains an operand-value to a particular instruction; for example, it may contain a floating point number as an operand to a multiplication instruction. The tag-value part of the token is broken up into the following fields:

TAG				
PORT	MAP	IP	PE	FP
1	7	24	10	22

- The **port** field is used to determine the input port of the instruction into which the token flows. For example, if the instruction is division, then the token could be either the divisor or the dividend, depending upon the port with which the token is labeled.

- The **map** field is used for interleaving large data structures across processors in order to more uniformly distribute memory references.

- The **pe** (processing element) field determines which processing element the token is currently in. This is sometimes used in some instructions which move tags into value slots in tokens, although on a given processing element, this field is redundant.

- The **ip** (instruction pointer) field is a pointer to an instruction in instruction memory into which the token is flowing. For example, the *ip* might point to an addition instruction.

- The **fp** (frame pointer) field is a pointer to the base of an activation frame in data memory where the token is to wait or match. The token will look into a particular offset, which is determined by the instruction, of this frame to find its partner. A new activation frame is allocated for each function invocation. All the tokens for any function invocation share an activation frame. In addition, each instruction is assigned an offset into its activation frame at compile time. In this way, different (i.e. parallel) invocations of the same function can co-exist: they simply have separate activation frames. Since each token is generated from an instruction, *each token knows exactly where in data memory to look for its partner.* This allows tokens to match in a single cycle, which is the principle advantage of ETS over previous dataflow architectures.

1.2.2 Instruction Fetch

An instruction is fetched from the instruction memory by the *ip* field of the token. The *ip* is just an absolute address into instruction memory. Instructions are broken up into the following fields:

INSTRUCTION			
OPCODE	*r*	PORT	*s*
10	10	1	11

- The **opcode** determines the operator (e.g. plus, minus, merge, switch, i-fetch, etc.) in addition to the number of outputs, how the outputs are to be queued, and how the operands are supposed to be matched. Since the opcode determines so many things, the 10 bits that we use to represent it are proving to be not enough. In the next machine, the opcode will have a larger field (12 bits).

- The **r** field is usually interpreted as the offset into the activation frame into which the token should look for its partner. If the token does not find its partner, it will insert itself into that slot.

  Some opcodes use the **r** fields for other purposes.

- The **port** field is the output port which the output token(s) is sent to on the next instruction. For example, if we executed the function $f(x) = (3 + x)/4$ The output of the addition of 3 and x would be sent to left port of the division instruction in the function, since $(3 + x)$ is the dividend.

- The **s** field is the offset from the current ip that the output instruction is located at. The **s** field is interpreted as a two's complement number because destination instructions can be located in instruction memory either before or after the current instruction. If the opcode specifies two outputs, the second output token is sent, by convention, to the following instruction, $ip + 1$. This convention has made compiling code for Monsoon somewhat awkward; in the next machine, there will be two s fields to represent the two offsets for two output destination instructions. The **s** field is also used for other purposes in certain opcodes.

1.2.3 Operand Matching

There are several different matching functions. For example, sometimes we have functions that have only one argument, such as *identity*. That token does not need to check for any other tokens. Sometimes, we have functions that take a constant as an input. For example, for the function $f(x) = x + 5$, the addition instruction will have only one incoming token – the 5 will be compiled as a constant into data memory. In these cases, the instruction will fire when the one token enters it since the other operand is a constant.

Most arithmetic instructions take two tokens and use a "normal" matching function. For those instructions, the first token will look at the presence bits at $fp + r$. If the presence bits indicate that the partner token is already in that slot, then the partner token is extracted, and the two tokens are sent to the next stage of the pipeline. If the presence bits for that slot indicate that the slot is empty, then the current token must be the first token to arrive. At that point, it inserts itself into the slot and changes the presence bits to indicate that it has arrived.

1.2.4 ALU and Create Tag or Bubble

If in the previous stage, the token discovered that the slot was empty, then the rest of the pipeline is bubbled, because there are not enough operands for the instruction.

If the tokens were sent to this stage from the previous stage, then two things happen in parallel: the tag or tags for the output token or tokens are formed, and secondly, the output value is calculated. If there are two output tokens, both tokens will have the same value. The output tags are usually constructed in the following manner:

- The **port** is the output-port indicated by the instruction. That is, it is the input port of the "next instruction".
- Currently, the **map** is the same as the last input token's map.
- The **pe** is the same as the last input token's pe.
- The **ip** for the first token is $ip + s$, and if there is a second token, its ip is by convention $ip + 1$.
- The **fp** is the same for most instructions, except certain instructions where the fp is either used for something totally unrelated to activation frames, or else used for entering or exiting function invocations.

1.2.5 Enqueue Token

If there was a bubble, then no tokens are enqueued unless one arrives from the network.

If there was one output token, then it is usually inserted directly into the pipeline, as was described in the Dequeue Token section.

If there were two output tokens, then one of them is inserted into the pipeline and one of them is stacked onto the current token queue.

There are some exceptions to these rules, but they are handled as special cases. Sometimes, there are tokens that are coming in from the network, and those tokens have priority over any tokens in the current processor in order to avoid deadlock in the network. Those tokens are always directly inserted into the pipeline.

1.2.6 Overview

Execution in the current system ends when the pipeline and the token queues are all empty.

Hopefully, the pipeline will be kept full most of the time. The ALU and Create Tag stage will be only full as much as the Operand Matching stage creates outputs. If all instructions took two operands, then the ALU would only be utilized half the time. In reality, almost half of the instructions take only one operand, so the ALU utilization is actually around 75%, if the pipeline is full.[2] In any case, it is not certain that ALU utilization is an important factor in determining the effectiveness of a parallel architecture.

Certain factors allow us to believe that the pipeline will be kept fairly full. The ETS pipeline is non-blocking; no instruction is dependent upon the execution of any other instruction, so there can be no conflict for data within the pipeline. Most programs have quite a bit of parallelism in them, so that if one section of the program is waiting for data, other parts of the program can still execute.[3] Data from GITA show that most programs end up using a lot of queue memory; this probably means that some of the tokens waiting in queues could be processed by the pipeline.

1.3 Example Program Compilation and Execution

Suppose that we would like to execute the following program:

```
def foo x =
  x * x + 2 * x + 7;
```

foo would compile to the following Monsoon code:

[2]Half of the instructions take only one operand because many of those instructions are actually identities. In real dataflow graphs, instructions often have more than two destinations. Since Monsoon only allows two destinations for each instruction, identities are used to "fanout" the outputs of some instructions.

[3]In fact, a big problem for dataflow is that it can exploit *too much* parallelism. Too much parallelism will use up too many resources too quickly, such as data memory and queue memory [6].

```
Code block FOO:
  IP: <OPCODE                                R => DEST:P>
-----------------------------------------------------
 438: <IDENTITY-M1                           0 =>  43D:0>
 439: <IDENTITY-M1                           0 =>  442:1>
 43A: <IDENTITY-M2                           0 =>  441:1>
 43B: <IDENTITY-M2                           0 =>  441:0>
 43C: <*R-L1                                4D =>  440:1>
 43D: <IDENTITY-M2                           0 =>  442:0>
 43E: <ADJUST-OFFSET-CHANGE-TAG-N1           2 =>    1:0>
 43F: <+R-L1                                4C =>  43E:1>
 440: <+R-N1                                 3 =>  43F:0>
 441: <*R-N1                                 1 =>  440:0>
 442: <ADJUST-OFFSET-CHANGE-TAG-N1           0 =>    0:0>
```

In the graphical representation of the dataflow machine graph (Figure 1.2) the part of the graph within the dotted lines is the actual code block; the other nodes of the graph are related to the function call and return.

Suppose we wanted to calculate **foo**(10.0). In the current "operating system", we would first write the argument, 10.0, to a special location in data memory.[4] The execution of **foo** would begin with the queueing of an initial token containing the address of **foo** and the address of the first free activation frame which is currently always the same location in data memory.[5]

This first token is copied by the first **identity** instruction and sent to the three nodes right above the **foo** code block – namely, the **change-tag** and the two **adjust-offset-change-tag** (**AOCT**) instructions.[6]

These three instructions are used to distribute information for the calling convention. The leftmost instruction, **change-tag** sends the address of the return instruction to

[4]The current "operating system" can only handle functions of one argument, but only because no one has bothered to implement anything more sophisticated. There is no inherent problem with functions of more than one argument. The current calling convention (taken from TTDA) supports functions with any number of arguments.

[5]Currently, activation frames are all the same size, and they are kept on a linked free list in data memory. Obviously, this will become more sophisticated as we learn how to write real operating systems for dataflow systems.

[6]Note that most of the identity instructions are used for "fanout" purposes.

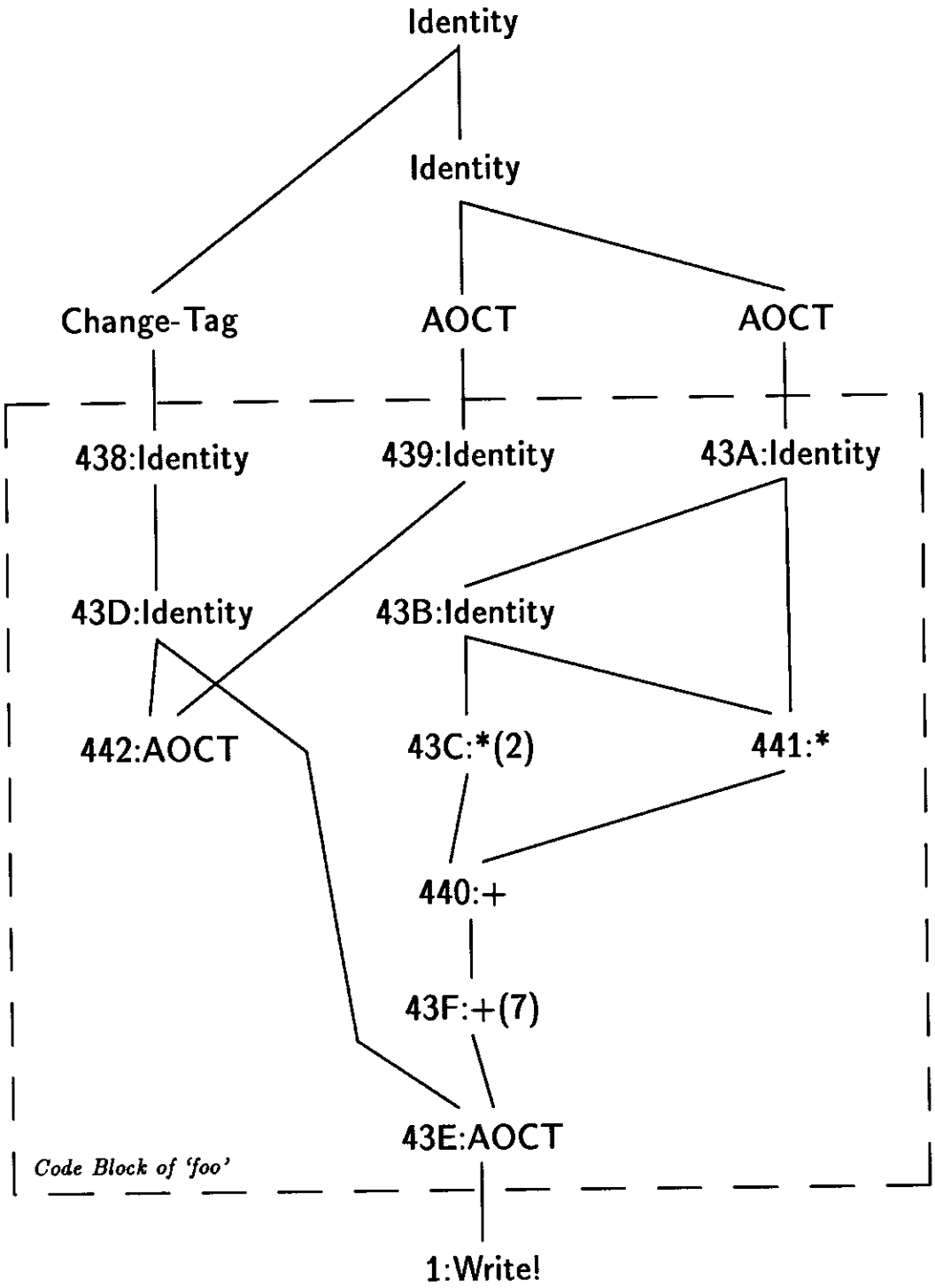

Figure 1.2: Graph for **foo** $= x^2 + 2x + 7$

the first instruction in the code block (which is an **identity** in this case). The return instruction is currently always the **write!** instruction. The middle instruction, **AOCT** sends the address of the argument list to the second instruction in the code block. In this case, there is no argument list, since we only have one argument. The rightmost instruction, another **AOCT** sends the value of the first (and only) argument to the third instruction in the code block. This convention is described in more detail in [12]; it is currently the same convention used for the TTDA.

The left and middle branches of the code block terminate. The **AOCT** instruction at offset `0xA` does not produce an output. Normally, the output of that branch is sent as a synchronization signal to the next code block, if necessary. We do not need the synchronization, so the **AOCT** will output to a **NOP** instruction, located at `0x0`. The return address will eventually find its way to the **AOCT** instruction at offset `0x6`.

On the right branch of the code block, the calculation of the function proceeds. At offset `0x9`, x^2 is calculated, and at offset `0x4`, $2x$ is calculated. These two are added at offset `0x8`, to produce $x^2 + 2x$ which is sent along to offset `0x7`. At this node, 7 is added, producing the final result, $x^2 + 2x + 7$. This result is sent to the right port of the **AOCT** at offset `0x6`. When the **AOCT** at `0x6` receives both inputs, it send off the result to the return address. The instruction at the return address is **write!**, which just writes the result to a pre-determined address in memory and set the presence bits to indicate that that location has been written to. Execution then halts, and the result can be read from that address.

Chapter 2

Why Build MINT?

With many new architectures, a simulator is built before the hardware is built in order to test the effectiveness of any architectural innovations. This was not the case with MINT. Since ETS was largely based upon work done on TTDA, Papadopoulos felt that most of the analysis of TTDA was valid for ETS.[1] MINT is intended to be a hardware debugging tool, although its development was parallel with the development of the hardware. With the addition of the microcode compiler [5], MINT is a true bit-level, microcodable Monsoon simulator.

2.1 The GITA Interpreter

The Computation Structures Group presently has a simulator for the TTDA architecture called GITA (Graph Interpreter for the Tagged Token Dataflow Architecture). Since TTDA is not an implemented architecture, GITA is not a realistic simulator; in fact, GITA is the only platform on which TTDA runs.[2] However, GITA is a very useful tool for evaluating algorithms on dataflow machines because it can take statistics. Most people who want to analyze algorithms do not care if the underlying architecture is somewhat unrealistic. In fact, they would prefer the effects of the operating system on the execution to be minimized, so that the characteristics of the algorithm, rather than the operating system, are observed.

[1] Papadopoulos did run some simulations on a slightly altered version of GITA, but no version of GITA has been a legitimate Monsoon simulator.

[2] Any architecture on which the only implementation is a network of 32 Lisp machines may reasonably be called "unrealistic".

GITA processes about 600-1000 tokens/second on an Explorer I Lisp Machine. TTDA tokens are a little more powerful than Monsoon tokens, since each token can output to multiple tokens, whereas Monsoon tokens can only output to a maximum of two tokens. Certain TTDA tokens are a lot more powerful because they are the equivalent of system calls. The bookeeping for those instructions is all done in Lisp, whereas in Monsoon, we are implementing an operating system on the machine. When MINT needs to make a system call, it will simulate the system call as part of the execution of the machine dataflow graph instead of directly implementing the call in Lisp.

The initial performance goal was to obtain comparable performance to GITA. It was known at the outset that it would be almost impossible to obtain the same performance or better performance because Monsoon programs will definitely require many more tokens to execute than TTDA programs. GITA has also been in constant use by the CSG since it was written about four years ago, so it has been highly optimized by many competent programmers. MINT would be written entirely by this author in a few months, from scratch.

2.2 Reasons for Building MINT

The rationales for MINT and GITA are different for one very important reason: Monsoon hardware. GITA was the only execution platform for the TTDA architecture. MINT is only to be used as an alternate execution platform for the ETS architecture. The Monsoon hardware will be about four or five orders of magnitude faster than MINT, which will practically eliminate the need for MINT as an experimental platform for large dataflow programs. The relationship between MINT and Monsoon is described in Figure 2.1.

All dataflow programs to be run on Monsoon or MINT will be written in the Id [4] programming language. Id is a high-level functional language with many novel features, chief among them being I-structures. People will seldom program dataflow machines in dataflow machine language, because it is too difficult. Even dataflow experts find it difficult to ascertain the correctness of simple dataflow machine graphs merely by "looking" at them. The Id compiler, which previously compiled TTDA graphs, has been modified to compile Monsoon machine graphs.

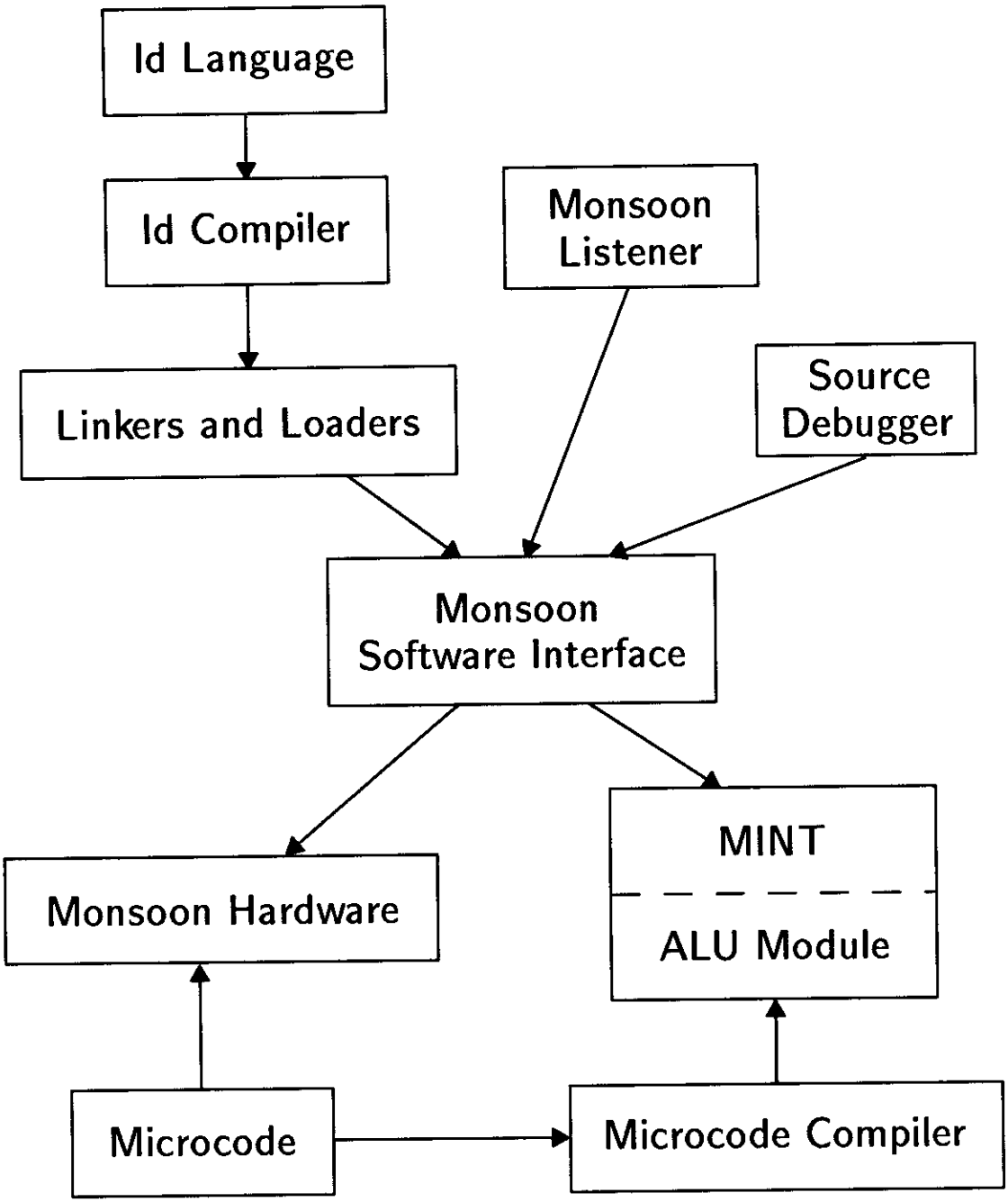

Figure 2.1: Where does MINT fit in?

The user front end will include the linker and the loader. ID programs are currently dynamically linked and loaded automatically after compilation, much as Lisp machines work. The front end will also consist of an execution environment, much like a Lisp Listener. The front end will direct either MINT or Monsoon to collect statistics and it will display those statistics. Monsoon contains some statistics registers which can be accessed by commands in the microcode. MINT has analogous "registers" which work like the hardware. In addition, Monsoon is capable of collecting other statistics which the hardware is not capable of collecting.

Both the Monsoon hardware and MINT can be modified by changing the microcode for the instructions. The microcode specifies several things: what ALU operation to execute, how many inputs, how many outputs, how to match the inputs, and how to generate addresses for the outputs. The microcode can be directly loaded into the hardware, and the microcode can be compiled into Lisp code [5], which can be inserted incrementally or totally into the Arithmetic, Matching, and Address Calculation (ALU) module of MINT.

MINT will not be used for running large experiments, unlike GITA and the MEF; it exists for different reasons. MINT will allow other research centers to perform experiments in dataflow without having to buy the expensive hardware. MINT will collect more statistics than is possible with the hardware. These statistics will provide important information about parallel algorithms and the Monsoon architecture itself. Of course, GITA served this purpose, but MINT promises to give a more accurate model of execution on an implementable architecture.

The current hardware is still not completely debugged, and with MINT, we have something with which to compare the hardware. MINT and Monsoon exhibit the same timing, and the same state in all of data memory, instruction memory and both token queues. Also, prototyping potential hardware enhancements will be much simpler in software than in hardware. If the hardware enhancements are simply writing new opcodes, the MINT microcode compiler will do that. All new instructions can be compared on MINT and the hardware.

2.3 Monsoon Software Interface Specifications

In the initial design phases of MINT, there were many complaints from group members about the programming style GITA is written in. Since GITA has been modified by many programmers, it is currently very "hacked-up". There was a question as to whether MINT should be based up the framework of GITA, and the response was quite negative. The consensus from the beginning was that MINT should be small, portable and have clearly specified interfaces.

These "clearly specified interfaces" evolved into the Monsoon Software Interface Specifications [13]. As indicated in Figure 2.1, these interfaces are used for both MINT and the hardware. There are more interfaces within MINT and between MINT and the microcode compiler, but the interfaces in the Specifications are to be used by external programs such as the Linker, and eventually, the Debugger. These interfaces must be supported by any simulator for Monsoon.

Chapter 3

Design Issues

In the initial design phases of MINT, several issues came up which influenced the design of MINT.

- MINT had to be reasonably fast, or else no one would use it.
- MINT had to be portable, so that it could be run on different systems, such as Symbolics, TI, and Sun Lisp.
- MINT had to be written with modification in mind. The Monsoon architecture is only an experimental prototype, so MINT had to be written cleanly enough so that other people could modify the code without a lot of effort.
- MINT was to be a bit-level simulator; this was a design decision which was heavily debated because GITA, the previous dataflow simulator, was not a bit-level simulator. This decision was the major force in the eventual design of MINT.
- MINT was *not* to support the Multi-Processor Emulation Facility (MEF), because of the existence of the Monsoon hardware.

3.1 Portability

Portability was important because MINT will be distributed to other sites, which may or may not have Lisp Machines. Therefore, we had the option of writing in either C

or Common Lisp. The compiler and all of the system software for Monsoon had been developed in Lisp, so we decided to implement in Lisp – speed was not that critical in the prototype simulator.[1]

Since the code had to be portable, we only used constructs in Common Lisp. For example, we did not use the Symbolics Flavor System, or any machine dependent function calls. We used few machine dependent optimizations, and each of those optimizations are conditionally compiled for each machine. There is always a Common Lisp implementation, which is portable, but slow.

3.2 Modularity

Modularity was an important design goal for MINT because several sections of MINT were to be experimented with immediately, which meant that those sections had to have clean, well-defined interfaces to the other modules. In addition, support for Gita had become somewhat tedious because it had grown as each user added features to it which he needed. These features were not always cleanly integrated into the structure of Gita; we decided that MINT would not have this downfall.

MINT is pretty much divided up as the hardware is. There are 5 modules:

- Memory
- Queues and Pipelines
- Matching, Arithmetic, and Address Calculation (ALU)
- Queueing and Control
- Statistics

Since MINT will be used for hardware prototyping, the microcode simulator for MINT is easily changeable. All of the changes to the microcode only affect the Matching, Arithmetic and Address Calculation module. The microcode does not have to be written

[1] In the future, a C version of the simulator may be implemented as a UROP project, now that there is already a design for the overall structure of MINT.

in Lisp. Derek Chiou is writing his bachelor's thesis on a Monsoon microcode to Lisp compiler [5]. This compiler will take a Monsoon microcode specification and compile it into Lisp functions which can be easily inserted into the ALU module of MINT. The microcode specifications are identical for the microcode compiler and for the Monsoon hardware.

The Queues and Pipeline module is separated out for easy experimention with different queueing strategies. For example, changes in the length of the pipeline are easily modified, and different types of queues (such as FIFO or random) could be easily implemented and inserted in place of the current module.

The Queueing and Control module can be easily modified to run different queueing strategies. Currently, there are several modes: single stepping, infinite processor emulation, and running until idle. With this module separated, finite processor emulation, and finite latency emulation can be easily implemented.

The modular breakdown of MINT lends to an easy implementation in other languages. Presumably, someone will eventually implement MINT in C, which may allow it to run several times faster than in Common Lisp.

3.3 Speed

Speed is not as important a criterion for MINT as it was for GITA, mainly because we have Monsoon, which will run programs about four or five *orders of magnitude* faster than MINT. We have no plans to implement MINT on the Multiprocessor Emulation Facility (MEF). The MEF is a high-speed network of 32 TI and Symbolics Lisp machines which can emulate computing on a multiprocessor environment. GITA could run on the MEF, mainly because GITA was the only "dataflow machine" that that we had until Monsoon. The MEF is extremely hard to program, and it is a very touchy piece of hardware. We decided that it was too much trouble to implement MINT on the MEF, especially since the hardware would still be three or four orders of magnitude faster than anything on the MEF.

However, we did not intend to implement a slow emulator merely because we had hardware. Other sites which would use MINT might not have the hardware. Therefore,

MINT would be the only thing (other than GITA) on which they could run dataflow experiments. Our goal was to have comparable speed to GITA, which processed about 1000 tokens/second on an Explorer I Lisp Machine. We achieved this goal, but unfortunately, the ETS model of computation presently requires about two to three times the number of tokens to execute identical programs. We believe that there is still room for improvement in MINT, but that we will never reach the same performance as GITA, because we have the added overhead of simulating an operating system, and because of the fanout necessary in the ETS model because each instruction is only capable of sending two other tokens. If a node in a dataflow graph requires more than two outputs, those additional outputs are implemented with additional identity nodes, which only take in one token, and output two tokens. Instruction counts on MINT indicated a much higher number of identities than in GITA.

3.4 Data Representation

The goal of bit-level simulation was at odds with our desire to implement a fast simulator. Aside from merely writing tight code, we had to keep two larger factors in mind in implementing MINT. First, we tried to avoid consing, in order to reduce the amount of time used by the garbage collector. We eventually wrote in a very imperative style, which is common in Lisp programs which require speed. Secondly, we tried to avoid too many type coercions, which were necessary because the operations on data were often on a different type of data than 64-bit integers. Since we were keeping a bit-level simulation, we were given the option of just implementing big arrays of bits which we could interpret as we pleased – essentially, just implementing the hardware exactly as it was designed. We eventually opted against such an implementation, because it would require too much type coercion. For example, because of the nature of Lisp, we could keep all data as the type it was last referenced as. If MINT executed a double-precision floating point multiplication, we could save the result in memory as a double-precision floating point number, since Lisp allows us to keep arrays of heterogenous objects. If we chose to reference an object as an integer, we would perform the coercion, and return the integer. All data was 64 bits long in data memory, whether it was a floating point number, unsigned integer, signed integer or tag.

In another issue of bit-level simulation, we decided not to implement the pipeline as

the hardware functions. We maintain timing consistency with the hardware by passing tokens through a FIFO queue as long as the pipeline, but all of the calculation for each token is done after the token exits this FIFO queue. It does not matter when the calculation for the token occurs, because the pipeline is a non-blocking, non-interlocking pipeline. None of the tokens can possibly depend upon the calculation of any other token because the dataflow model does not care what order the tokens are processed in. By forsaking this level of accuracy in the simulation, we were more easily able to abstract the function of each opcode in Monsoon, since the function of each opcode was executed all in one Lisp function.[2]

The token queues were implemented in a similar manner to the memory. Although the data in the queues was always kept as token structures, which were implemented as Lisp structures, we had the ability to reference the queue as 128 bit unsigned integers, if we pleased. The FIFO queue for the pipeline was represented in the same manner. More details of the data representation are discussed in the next chapter.

[2]After the implementation of MINT, we realized that it may be possible to extract all of the state of the pipeline from the opcode functions, if necessary. In normal simulation, this would only slow down the simulator; however, we would not need to extract this information in normal simulation. In single stepping mode, this may be useful in comparisons with the Monsoon hardware

Chapter 4

Program Structure

4.1 Overview

Each of the modules of MINT corresponds to a certain section or sections of the Monsoon architecture (see Figure 4.2). The Memory module simulates the Monsoon memory. The Queues module simulates the Monsoon queues. The Matching, Arithmetic, and Address Calculation module simulates the stages in the pipeline which perform matching, arithmetic and address calculation. The Queueing and Control module controls the movement of data between all of the other modules. It dequeues the tokens, fetches the instructions, sends the tokens to the Matching, Arithmetic and Address Calculation module. That module returns either zero, one, or two tokens and information about how to enqueue those tokens to the Queueing and Control module. Then the Queueing and Control module enqueues the tokens, and loops back to the dequeue stage.

MINT does not simulate the inside of the pipeline. Instead, all of the work of the pipeline is done in the Matching, Arithmetic, and Address Calculation module. The timing for the pipeline is simulated by passing the enqueued tokens through a circular buffer as long as the pipeline. All of the calculation for the pipeline can occur either before insertion into, or after removal from the circular buffer. We chose to do the calculations after the removal because the calculations increased the amount of data; before calculation, we have one token – after calculations, we could have zero, one, or two tokens.

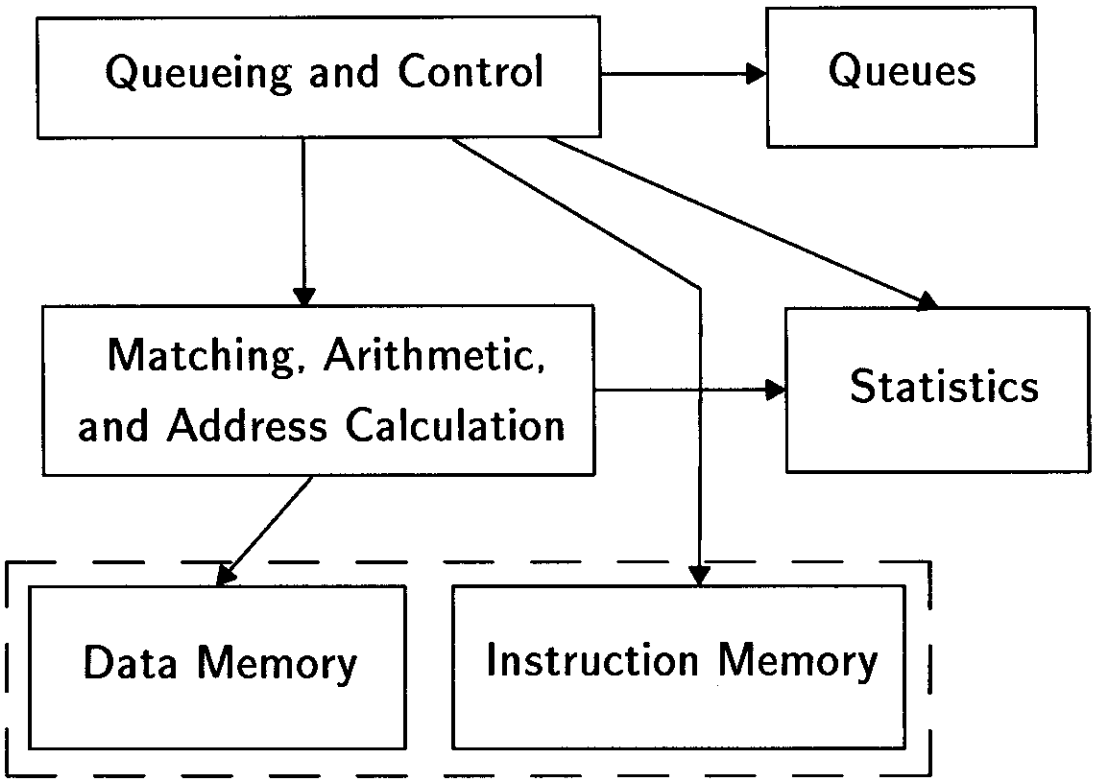

Figure 4.1: MINT Module Dependencies

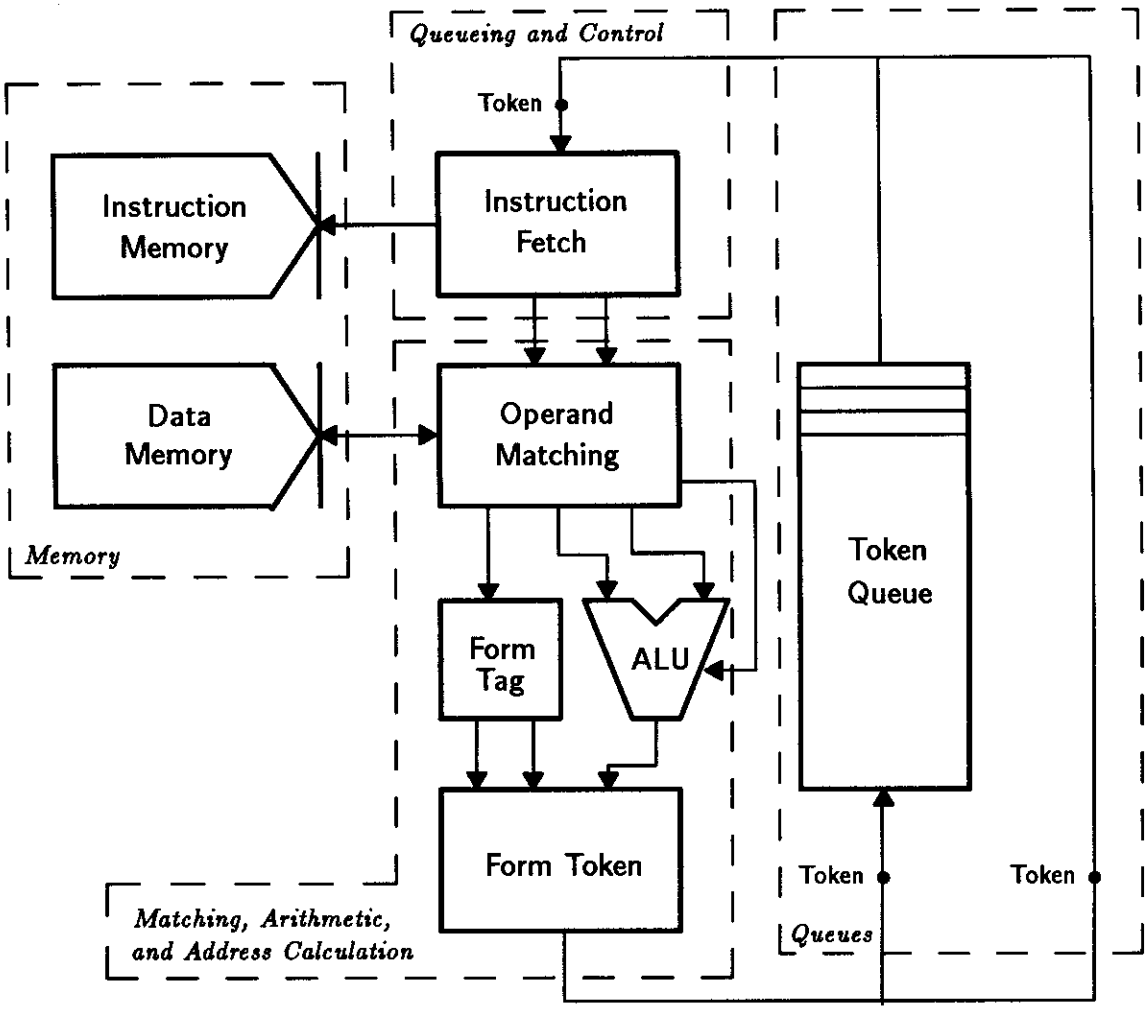

Figure 4.2: MINT Module Distribution

4.2 Memory

Memory seems to be a trivial issue, since we can implement memory as a big array of numbers, just like it is in the hardware. Unfortunately, we come across some problems when we implement memory in this way. First of all, data memory is 64 bits wide in Monsoon.[1] In most implementations of Lisp, a big array of 64-bit integers is actually a big array of pointers to bignums, which must be consed up and garbage collected. Initially, we considered implementing memory as two big arrays of 32 bit quantities, but we quickly realized that any arithmetic would probably eventually require conversion of the 32-bit quantities to bignums first.

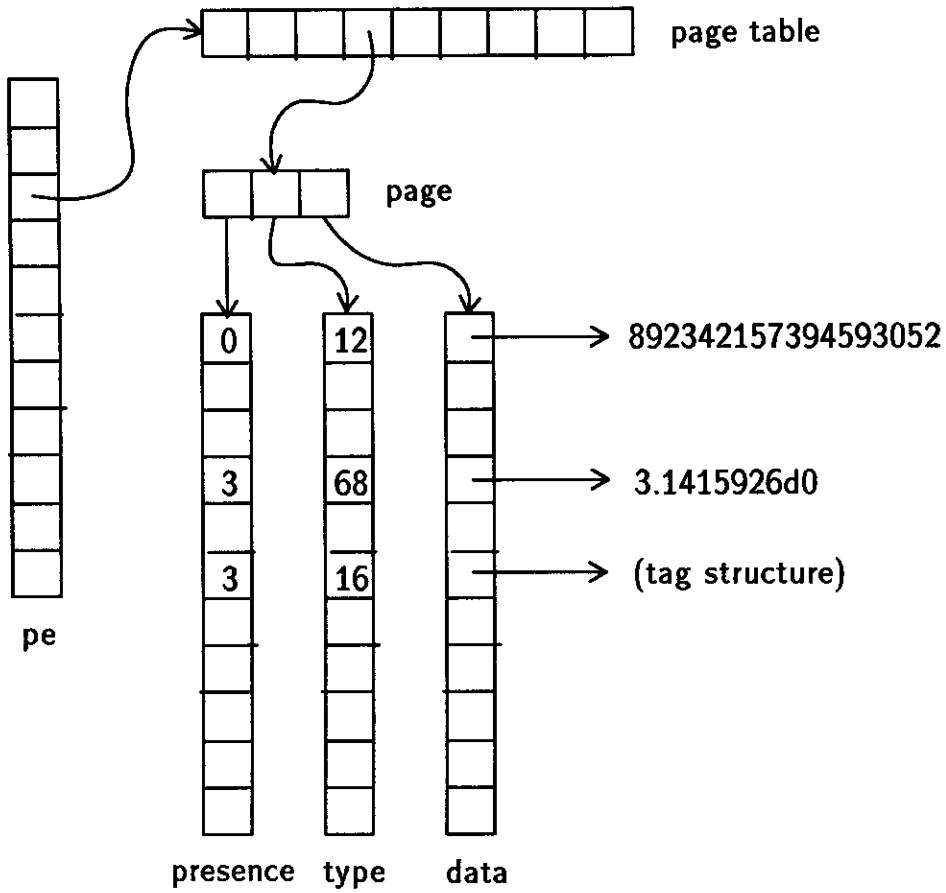

Figure 4.3: MINT Data Memory

Eventually, we decided that we could not avoid consing bignums. However, we knew that we would have to do a lot of type coercion if we kept all the data in the form of

[1] Actually, memory is 64 bits of data, 8 bits of type and 2 bits of presence – 74 bits wide.

bignums. Most of the math is done with IEEE double-precision (64-bit) floating-point numbers. Whenever MINT would do a floating-point operation, both operands would have to be converted to double-floats, the operation would be executed, and the answer would have to be converted back to a bignum. Since Common Lisp allows us the luxury of arrays with heterogeneous types, we opted fill the slots with the most convenient objects. Each slot would be left as the type of the last reference to that slot. Each memory reference would have a type associated with it. For example, we can read any slot in data memory as one of the following types: unsigned integer, signed integer, double-float, or tag. In instruction memory, which is 32-bits wide, we can read any slot as either an unsigned integer or an instruction. All memory writes leave the object as the type that it was.

Note that this does not affect the state of the machine to any user viewing this memory. He cannot tell the difference between the hardware and the software, even though the software keeps the data in this strange format. If he wants to read any slot as an unsigned integer, he can just specify it as such. If the slot in the array actually contained an object of another type, then the read function automatically coerces the object for him, in addition to writing the new object back into the slot, on the assumption that the new type is going to be the type that the object is *usually* referred to. This has been tested and proven more effective than just leaving the object as it was originally written.

We had another problem with the memory. The address space was 24 bits wide. This meant that a straightforward representation of memory would require an array that was 16 megawords big. In addition, since data memory also contains type (8 bits) and presence (2 bits) information, we would need another two large arrays. Consing up three 16 megaword arrays was out of the question. We had the option of simulating just part of memory. However, we decided that that was not enough. Eventually, we decided to implement a "demand-paged" memory. Each memory reference would go through a "page table". On each memory reference, the top 14 bits were used as an index into the page table. If the entry in the page table were empty, then we would create that page. Otherwise, we would use the page in the page table slot. The remaining 10 bits were used to index into the page. In this way, we only needed a 16 kiloword page table, and each newly referenced page was 1 kiloword big.

Unfortunately, since data memory contains presence, type and data fields, and since

we supported a multiple processor memory model, each memory reference required four array references, in addition to type coercion and automatic page creation if necessary. Instruction memory only requires three memory references, and it is not clear yet whether we need 24 bits of instruction memory to run any program on MINT. Any program that large would take an unreasonably long time to execute.

We do not suggest that any program run on MINT would need to use all 24 bits of data memory. However, the alternative to not supporting the all 24 bits is to support less than 24 bits. If a user runs a program which needs more memory than we support, what shall we do? We must enlarge the memory enough to run that program. It is better to support the memory now than allow some user to crash sometime in the future. This paging scheme also only allocates as much memory as the user requests, aside from the overhead of the page table.[2]

The instruction memory is exactly the same as the data memory, except that the instruction memory does not have a structure containing data, type and presence, since instruction memory does not have type and presence information. All the slots in instruction memory are represented as instruction structures, since instruction memory is rarely referenced as anything else except as instructions.

Since the memory is constructed in this somewhat odd fashion, several optimizations became clear once we began to implement the Matching, Arithmetic and Address Calculation module. Many instructions need to read and write to several fields at one address. For example, if a token uses a normal matching function, it will read the presence bits at the specified location. If the presence bits indicate the slot is empty, then the token inserts its data into that slot and writes the presence bits to indicate that the slot is full. If the presence bits indicate that the slot is full, then we want the value of the slot, and we want to write the presence bits to indicate that the slot is empty. If we had implemented the memory with a flat array, then each of the steps could be done independently. However, since we have a heirarchical memory structure, we would be doing redundant array references if we referenced each field separately (i.e. first reference the presence bits, then reference the data, then reference the presence bits again). Since this is a common matching function, we designed a memory reference function, **rw-dm-data**, which takes an address and a value as arguments; if the slot is empty, the value is written

[2]However, most of the programs we have run so far actually use less memory than the page table has (16 KWords).

and the presence field is set to full and *nil* is returned; if the slot if full, then the presence bits are set to empty, and the value of the slot is returned. Functions such as this are used throughout the ALU module, and the Monsoon microcode compiler generates code which uses these function.

Since MINT is very modular, it would be a trivial task to implement a faster, but more limited version of the memory module. The module need only support the same memory instructions specified by the Monsoon Software Interface Specifications, and the following instructions implemented for use with the Monsoon microcode compiler:

`read-dm-data` *pe address* [*Function*]
`write-dm-data` *pe address value* [*Function*]
`write-dm-dtp` *pe address data type presence* [*Function*]

These two functions do not coerce types. Any object can be written or read as its current type. Note that the *type* argument to `write-dm-dtp` does not affect the type of the data.

`write-presence-and-read-dm-data` *pe address presence* [*Function*]
`write-presence-and-read-dm-data-as-bits` *pe address presence* [*Function*]
`write-presence-and-read-dm-data-as-integer` *pe address presence* [*Function*]
`write-presence-and-read-dm-data-as-float` *pe address presence* [*Function*]
`write-presence-and-read-dm-data-as-tag` *pe address presence* [*Function*]

These five functions write the presence as *presence*, and reads the data as the indicated type.

`rw-dm-data` *pe address value* [*Function*]
`rw-dm-data-as-integer` *pe address value* [*Function*]
`rw-dm-data-as-bits` *pe address value* [*Function*]
`rw-dm-data-as-float` *pe address value* [*Function*]
`rw-dm-data-as-tag` *pe address value* [*Function*]

Each function will write the value to the specified address if the presence bits indicate the slot is empty. The presence bits will be changed to indicate the slot if full and *nil* is returned. If the slot is full, then the value in the slot is returned as the type indicated by the function name.

4.3 Queues and Pipelines

Both queues and pipelines are just arrays of pointers to tokens structures. Token structures are defined as follows:

```
(defstruct (token :conc-name)
  (tag-type 0)
  (port 0)
  (map 0)
  (pe 0)
  (ip 0)
  (fp 0)
  (value-type 0)
  (value-value 0))
```

Although the token is usually abstractly considered as the four-tuple of tag-type, tag-value, value-type, and value-value, we chose to explode the tag-value slots because keeping the tag-value as one slot containing a tag-value structure would only add to interpretation costs. *Every* time we dequeue a token from the pipeline, we will *always* require every slot of the tag portion of the token. Therefore, we chose to eliminate the one extra reference, and the extra consing involved with keeping the abstraction.

The tag-type and value-type slots are not currently used, but they would eventually presumably contain 8-bit quantities. The value-value slot will contain some sort of value. As described in the **Memory** section, the types of values are as follows: unsigned integers, signed integers, 64-bit floating point numbers, and tags. If the type is already correct, then coercion will be unnecessary. As was already stated, coercion is usually unnecessary.

The token queues are just empty art-q arrays and a pointer; they are just stacks. When a token is enqueued, if that slot in the array is empty (i.e. nil), then a new token will be consed up and filled with the contents of the enqueued token. That new token will be inserted into the empty slot, and the queue pointer will be bumped up. If the current slot in the array already contains a token structure (which is invalid, having already been dequeued) then the slots of that structure are just filled with the values of the enqueued token. In this way, we only cons up the number of tokens necessary when the token queue

is most full, and we avoid discarding token structures which will just give more work to the garbage collector.

The pipeline is just an art-q array and a pointer to a slot in that array. The current pipeline is 8 stages deep, so it is implemented as in Figure 4.4.

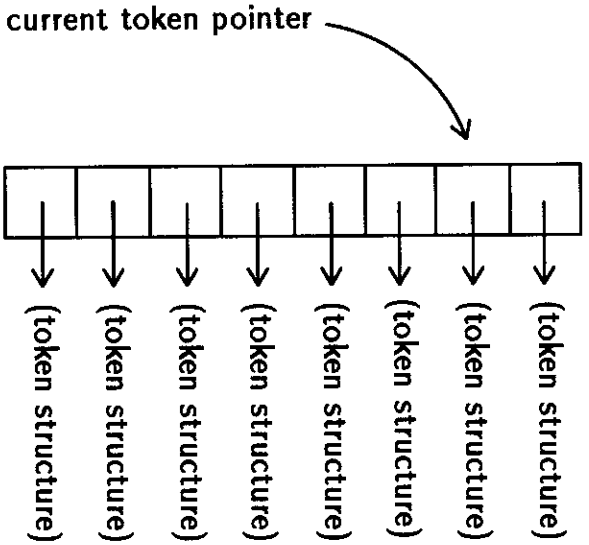

Figure 4.4: MINT Pipeline Representation

The pipeline pointer points to the current token to executed. First, that token is dequeued. Then that slot does not contain anything useful. After the token is operated upon (i.e. sent through the pseudo-pipeline) then a new token will be enqueued into the pipeline. That token will take up the same slot that the old token just exited. We then bump the pointer. When the pointer reaches the end of the array, it wraps around to the beginning.

The pipeline is always filled with token structures. When a new token is enqueued into the pipeline, we just mutate the slots of the old token. The old values of that token have already been dequeued, and are no longer needed. In this way, we avoid consing up new token structures which will just have to be garbage collected in the future.

Another advantage of this implemenation is that we can dequeue tokens from the queue directly into the pipeline by swapping the pointers to the tokens. This takes far less steps than copying the contents of each field of the queue token into the pipeline

token. Unfortunately, this tends to spread the tokens out in memory, and destroys locality. These little tokens structures could be spread out all over memory. Pointers to these tokens will be shuffled about by the swapping of pointers.

An alternative implementation is to use a flat array. We still want to use the same fields, but we can spread out the fields across each slot in the array. In this way, we preserve locality, but we must cons up all of the memory we need for the queues when be begin computation.[3]

4.4 Matching, Arithmetic, and Address Calculation

This module is the heart of the simulator. Most of the pipeline operations are done in this module. In Figure 4.2, there are two arrows entering the box representing this module, and one arrow leaving the box. The two arrows entering the box are the data from the instruction and the data from the token. The one arrow leaving the box is the output token or tokens. These tokens also carry with them information on how they should be queued.

We chose to implement this module as a set of Lisp functions; each opcode is given a separate Lisp function, since it is the opcode which determines the operations within the box. Each Lisp function takes the same arguments (just the sum of the number of fields in a token and the number of fields in an instruction) and returns either zero, one or two tokens, and instructions on how to enqueue those tokens. The arguments to the Lisp function are the slots of the token, and the slots of the instruction. The actual enqueueing of the output tokens is done by the Queueing and Control module.

Each opcode has a different matching criterion, function, output destination computation, and number of outputs. This is reflected in the Lisp code. For instance, a hand-coded opcode is shown below:

[3]We currently do not automatically grow the queues when they become full, although this is an obvious feature that should be implemented, for the same reason we implemented the paged memory abstaction.

```
(defun +-n1 (tag-type input-port map pe ip ; token fields
             fp value-type value-value     ; token fields
             r output-port s)   ; instruction fields
   (let ((wm-val (rw-dm-data pe (+ fp r) value-value)))
      (when wm-val
        (let ((result (+ (coerce-to-float value-value)
                         (coerce-to-float wm-val))))
          (values :enqueue-hp-queue ; how to queue
                  tag-type          ;
                  output-port       ; specified by instruction
                  map               ; not currently used
                  pe                ; not currently used
                  (+ ip s)          ; standard output
                  fp                ; within the same frame
                  value-type        ;
                  result)))))       ; x + y
```

Note that this opcode uses the "standard" matching function – each token looks for its partner at $fp+r$. When a token finds its partner, this function adds the values of the extracted data and the data carried by the current token. This is then returned as the data for the output token. The output token is sent to the "standard" location of $ip+s$.

The values returns by the Lisp function are the slots of the output token, and instructions on how to queue that one token. The `:enqueue-hp-queue` keyword tells the *run* function to enqueue this token in the high priority queue.

4.5 Queueing and Control

This module is the brains of the whole operation. It is the top level loop which controls the rest of the modules. Basically, it just follows the ETS pipeline diagram. Depending upon how we wish to queue tokens and move data, the top level function, `run` will preform accordingly. For example, `run` may only execute one instruction cycle in single-stepping mode, or it may just run until the queues are empty in normal mode, or it may gather statistics for infinite processor mode.

Normal run has the following pseudo-code.

1. If the pipeline and queue is empty, then halt.
2. Dequeue a token from a queue.
3. Fetch the instruction indicated by the token.
4. Call the function corresponding to the opcode. The arguments to the function are the fields from the token and the instruction.
5. Take the tokens and the instructions returned from the previous function call and enqueue the tokens according to the instructions.
6. Loop back to the beginning.

Infinite processor run has the following pseudo-code.

1. If the pipeline and both queues are empty, then halt.
2. If the current queue and the pipeline are empty, then swap the queues and write the statistics to registers.
3. Dequeue a token from a queue.
4. Fetch the instruction indicated by the token.
5. Call the function corresponding to the opcode. The arguments to the function are the fields from the token and the instruction.
6. Take the tokens and the instructions returned from the previous function call and enqueue the tokens according to the instructions *in the other queue.*
7. Loop back to the beginning.

Note that the only difference between the two *run*'s was the use of the statistics registers and the swapping of queues.

4.6 Statistics

The statistics registers are currently implemented exactly as the Monsoon Software Interface Specifications require. They are a bank of arrays which can be written to at appropriate points in the code. For instance, if we wanted to do an instruction mix, we could compile the microcode specifications so that certain opcodes incremented certain statistics registers every time they were called. If we want to do a parallelism profile, a certain register can be incremented on every step through the top-level run function. These statistics registers would be read by the debugger or the Monsoon Listener at appropriate points in the execution.

Chapter 5

Conclusions

MINT has met the performance specifications that we set for it at the beginning of the design stage. It will simulate Monsoon at a bit-level, although the internal representation of MINT is not necessarily all bits. All of the timing for MINT is identical to Monsoon, making it suitable for debugging Monsoon hardware. And MINT is comparable in speed to Gita, in terms of tokens processed per second. However, since MINT will execute about two and a half to three times as many token, MINT will execute most programs in about 50

In the end, the decision to implement the simulation at the bit-level turned out to be an important one in the utility of MINT. The author firmly believes that researchers only interested in collecting statistical data on dataflow programs should adopt the approach of Muryanto and Tan [9] [11]. Muryanto and Tan chose to compile dataflow graphs directly into sequential assembly code; preliminary analysis of the speed of this approach suggests at least an order of magnitude speed increase over an interpreter approach such as MINT. Since MINT is an exactly simulates all state and timing of Monsoon hardware, it will find applications in the debugging of this hardware, in addition to debugging of multiple processor Monsoon hardware in the near future.

There are several obvious extensions to MINT which should be considered in the near future. First of all, MINT should simulate multiple processors. All of the modules of MINT already individually simulate multiple processors – for example, memory references must specifiy the processor which the reference is pointing to. However, there is no network simulator module. Once this is accomplished, an I-structure board module

should be written. I-structure boards are separate components of dataflow computers which are attached to the network, and serve all I-structure requests.

MINT should be written in C, so that it can be transported to even more systems than it will be available in now. The C version of MINT should be several times faster than the Lisp version, which will make MINT more attractive to researchers who do not want to buy the Monsoon hardware. If MINT is ported to C, then perhaps some sort of network protocol can be devised for parallel execution of MINT across several different kinds of computers. Although we decided not to support the MEF, extending MINT to run on the MEF would probably not be very difficult.

Bibliography

[1] Arvind, S. A. Brobst, and G. K. Maa. Evaluation of the MIT Tagged-Token Dataflow Project. Technical Report CSG Memo 281, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, 1988.

[2] Arvind and D. E. Culler. Dataflow Architectures. In *Annual Reviews in Computer Science*, volume 1, pages 225–253. Annual Reviews Inc., Palo Alto, CA, 1986.

[3] Arvind and R. S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. In *Proceedings of the PARLE Conference, Eindhoven, The Netherlands. (Lecture Notes in Computer Science Volume 259)*. Springer-Verlag, June 15–19 1987.

[4] Arvind, R. S. Nikhil, and K. K. Pingali. Id Nouveau Reference Manual, Part II: Semantics. Technical report, Computation Structures Group, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, April 1987.

[5] D. T. Chiou. A Reverse Compiler: Monsoon Dataflow Microcode to Common Lisp. Technical report, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1989.

[6] D. E. Culler. *Effective Dataflow Execution of Scientific Programs.* PhD thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 77 Massachusetts Avenue, Cambridge, MA 02139, December 1988 (expected).

[7] J. R. Gurd, C.C. Kirkham, and I. Watson. The Manchester Prototype Dataflow Computer. *Communications of the ACM*, 28(1):34–52, January 1985.

[8] K. Hiraki, S. Sekiguchi, and T. Shimada. System Architecture of a Dataflow Supercomputer. Technical report, Computer Systems Division, Electrotechnical Laboratory, 1-1-4 Umezono, Sakura-mura, Niihari-gun, Ibaraki, 305, Japan, 1987.

[9] L. Muryanto. A Translator from ETS Dataflow Graphs into RISC Code. Technical report, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1989.

[10] G. M. Papadopoulos. Implementation of a General Purpose Dataflow Multiprocessor. Technical Report CSG Memo 432, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, December 1988.

[11] P. Tan. A Translator from RISC Code to MC68020. Technical report, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1989.

[12] K. R. Traub. A Compiler for the MIT Tagged-Token Dataflow Architecture. Technical Report LCS TR–370, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, August 1986.

[13] K. R. Traub. Monsoon Software Interface Specifications. Technical Report CSG 296, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, May 1989.