LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

A Reverse Compiler: Monsoon Dataflow Microcode to Common Lisp

Computation Structures Group Memo 299
June 1989

Derek Chiou

Submitted to the Depaıtment of Electrical Engineering and Computer Science 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

A Reverse Compiler: Monsoon Dataflow Microcode to Common Lisp

by

Derek Chiou

Submitted to the Department of Electrical Engineering and Computer Science
on May 10, 1989
in partial fulfillment of the requirements for the degree of
Bachelor of Science in Electrical Engineering and Computer Science

Abstract

Monsoon, a dataflow processor, has recently been built at the Laboratory for Computer Science at MIT. To facilitate experimentation, Monsoon employs a downloadable microcode to describe its instruction set. Since we wish to accurately simulate the processor, regardless of its current microcode, the concept of a microcode compiler was developed.

The goal of the thesis was to design and code a compiler that accepts a Monsoon microcode specification and a Monsoon instruction set specification as inputs and translates them into Common Lisp for use within an existing Monsoon simulator. The primary objective of the compiler was to produce code of comparable efficiency to hand-coded routines. Secondary considerations were human-readability and code size. Thus, the identical microcode used to drive the actual hardware is also compiled for the simulator, with obvious benefits of hardware – simulator consistency.

A compiler has been written, and it produces Common Lisp code that satisfy the above goals. It is flexible enough to adapt to any foreseeable microcode changes. The compiler is written in Common Lisp.

Thesis Supervisor: Dr. Gregory Michael Papadopoulos

Title: Project Manager

Acknowledgments

Greg Papdopoulos was a great thesis advisor. His careful reading and his excellent advice were instrumental in the writing of my thesis.

Jamey Hicks, Ken Steele, Ken Traub, and Jonathan Young were the ones I turned to for help with this thesis. I'm grateful for their patience with me, for their help when I needed it, and for their guidance when I didn't.

Steve Brobst got me started in the world of computer architectures. It is his example that I try to follow.

Professor Arvind permitted me to learn from him and his group. It is this knowledge that will be the base of my future.

My friends, especially Angelina who walked beside me through our neglected classes, the Quint who didn't forget me though I seemed to have forgotten them, and Ning who was always there to talk to me, gave me the support I needed.

And, last but not least, my family, of course.

To my parents,
Jiunn P. and Rita Chiou

Contents

List of Figures

Chapter 1

Introduction

The Computation Structures Group of the Laboratory for Computer Science recently assembled a prototype of a Monsoon Dataflow processor as defined by Papadopoulos[6]. Monsoon was designed with an easily modifiable instruction set[10] – trivial changes to data loaded into the processor, which we call microcode, can achieve a broad spectrum of processor behavior. Monsoon achieves this flexibility by dynamically decoding instructions along with data state into microcode that controls its pipeline stages. Thus, the process performed at each stage depends on the data being processed and the microcode associated with that data. Since the processor was designed to invite instruction set experimentation, the group is doing just that. This development is being done both on the Monsoon hardware and a Monsoon Interpreter (MINT) which was written by Shaw[7]. This thesis aids in that experimentation process by compiling microcode intended for the Monsoon hardware into efficient Common Lisp suitable for use in MINT.

The way dataflow programs are executed is shown in Figure 1.1 and is described below. The programmer writes the program in a high-level language called Id[5]. The Id program-to-graph compiler[9] then compiles the program into a dataflow graph. Once the graph is created, execution can take one of two paths. Either the Monsoon hardware or MINT could run the program. There are advantages and disadvantages to both execution systems. The hardware processes tokens close to four orders of magnitude faster than the simulator but does not provide sophisticated debugging and statistical tools. Another disadvantage is that only one copy of the hardware exists – thus, the number of people that can perform research is very limited. MINT, though very slow, will run under any Common Lisp environment and will include debugging and analysis tools.

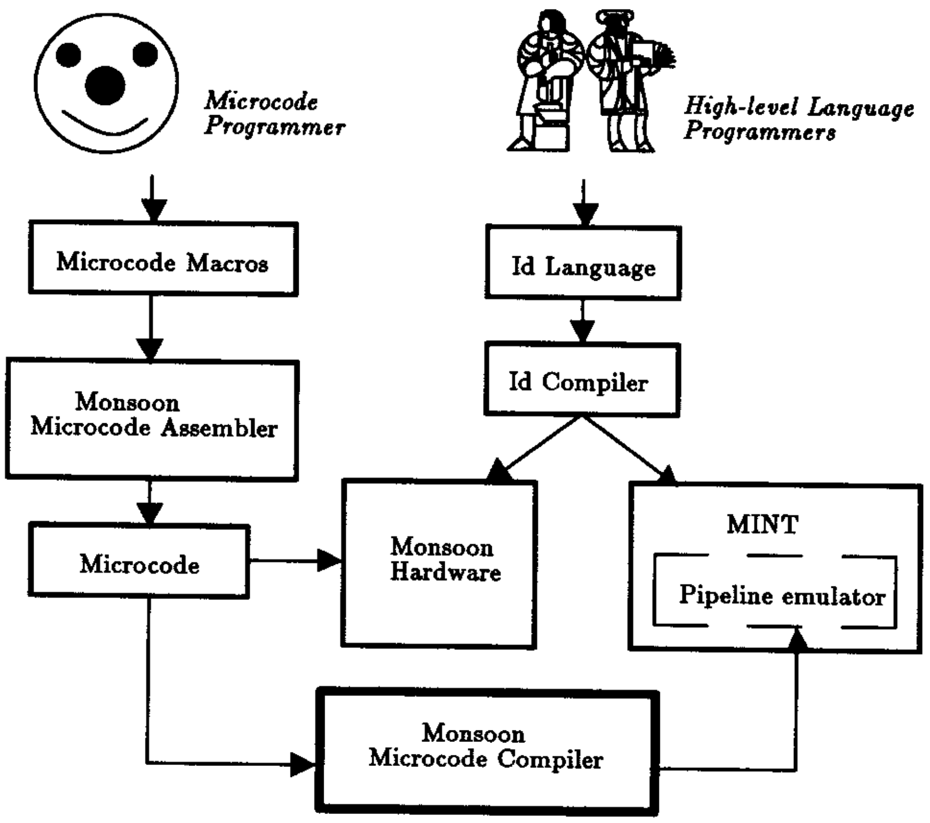


Figure 1.1: Monsoon and its Programming Environment

Since the raison-d'être of the simulator is to emulate the hardware, any changes in the hardware must also be made in the software. At this point in the hardware refinement, the microcode, and thus the instruction set, is the only part that will change. Since new microcode can be loaded into the hardware at will, a method to load new microcode into the simulator should be provided. A simple solution to is to build a microcode simulator that interprets the microcode. Interpretation, however, would be unacceptably slow. Our solution, the goal of this thesis, is a compiler that takes the microcode and uses it to create efficient Common Lisp code. This compiler is rather unique in the fact that its source language is low-level microcode and the destination language is high-level Common Lisp. Since we desired efficient code, a direct mapping of microcode to Lisp is not sufficient. We identified many opportunities for optimization, most of which were incorporated into the compiler.

The compiler was also designed to be general enough to make changes in the microcode easy to make in the compiler. This generality is achieved by specifying decode tables as parameters. Changes to the microcode structure itself, say by a hardware modification, can be reflected in the compiler by changing the decode tables. Note that if the structure of the tables changes, modifications to the compiler code will probably have to be made. The compiler assumes that the basic pipeline structure will remain constant.

With the compiler in place between the microcode and the simulator, our system is complete. When the microcode of the hardware is changed, the simulator is changed by a simple compilation. The compiler will encourage experimentation with microcode by making it easy to simulate.

Chapter 2

The Monsoon Processing Element

2.1 Dataflow Background

Dataflow is a relatively old idea in the field of computer architectures. It was first proposed by Dennis and Misunas[2] and later refined by a number of projects including the Manchester Machine[3], Sigma-1[8], and TTDA[1]. The basic concept behind dataflow is that execution is *data-driven* rather than instruction-driven. An instruction executes only after all the data it requires have been received. Conventional computers, on the other hand, execute instructions in a specific sequential order with little run-time regard to data dependencies. It is evident that dataflow computers have the potential for exploiting large amounts of parallelism inherent in many applications, and that data-driven execution readily extends to parallel architectures.

The standard abstraction of a tagged-token dataflow machine was developed independently by groups at Manchester University of Manchester, England, at the University of California, Irvine, and later refined by the Monsoon processor, where associative memory has been replaced by explicit memory storage techniques. Data travels through a dataflow machine in packages called *tokens* consisting of data along with control information, called *tags*, that encode such data as the destination instruction. The advantage of using a tag is that each instruction is synchronized on its own – it requires no external information. Thus, it is easier to keep processors busy using tokens with tags than tokens without tags.

The basic operation of a tagged-token dataflow processor is described. A token enters a processor and checks for its partner, presumably by checking for equal tags. If the partner is present, it is fetched and the instruction is executed. Obviously, if the instruction is unary, such as NOT, the processor does not need to check for a partner. The part of the machine that checks for partners is generally called the *waiting/matching* area. Newly calculated data is encapsuled into tokens which are sent back into the system. A dataflow processor cycles through tokens until an answer is produced.

Though the dataflow concept is an intuitive one, efficient ways of implementing the concept are not so intuitive. Some problems with dataflow are the necessity of immutable data-structures to prevent out-of-order errors, large, fast waiting/matching sections, garbage collection of objects being referenced in an out-of-order fashion, deadlock resolution, and so on. Efficient dataflow processors are thought to require special hardware to support the dynamic scheduling associated with the computation model. Though implementation challenges exist, however, the promise of being able to exploit nearly all parallelism within an algorithm is too good to pass up.

A few dataflow machines have been built, the most notable being the Sigma-1[8] and the Manchester Dataflow Machine[3]. Various design details, however, hamper these machines. A major detail is the implementation of the waiting/matching area. The simplest waiting/matching area has fully-associative memory so that partner searches take unit time. Fully-associative memory is much too expensive for any reasonably sized waiting/matching area, so cumbersome hashing techniques are generally used. Papadopoulos[6] provides an elegant solution to the matching problem in the implementation of Monsoon, a pipelined dataflow processor.

2.2 Monsoon: a Dataflow Processor

Monsoon is a fully pipelined dataflow processor – thus, none of its stages can take more than unit time to execute (there are a few exceptions, but they rarely occur.) Monsoon's basic architecture, some of its unique features, and an overview of its microcode decoding are described in this section. Much of this information was taken from Papadopoulos[6].

The basic stages of Monsoon, as shown in Figure 2.1, are *instruction fetch*, *effective address generation*, *fetch presence bits*, *operand fetch and/or store*, *ALU/FPU and next address generation*, and *form token*. The operation occurring in any stage of the pipeline is completely independent of the operations occurring in the rest of the pipeline. Each

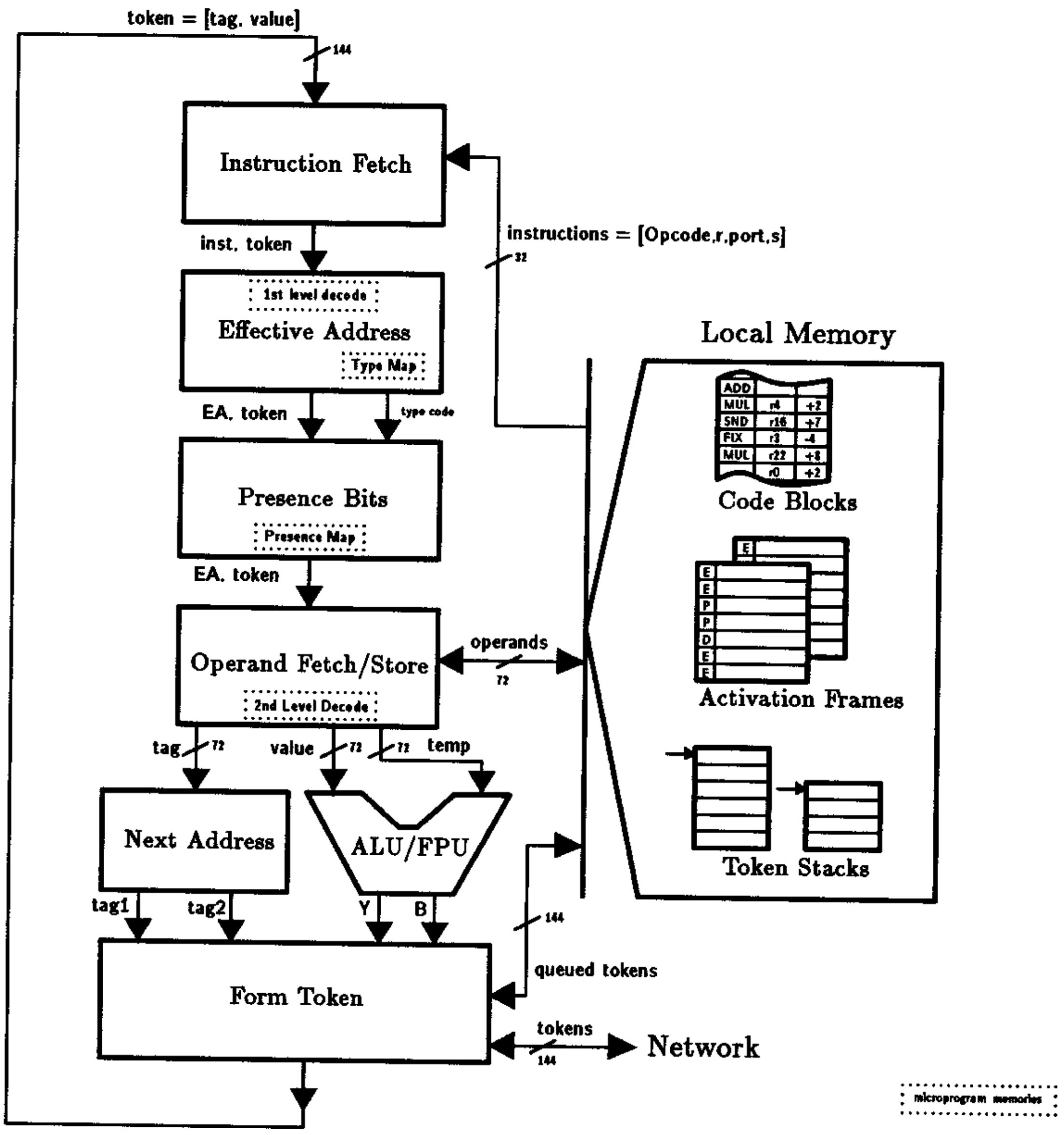


Figure 2.1: Monsoon Processing Pipeline Overview

stage is also non-blocking except for some exceptions due to read/writes to memory and long floating-point operations. Though the pipeline can block to permit long operations to finish, the pipeline never needs to be flushed since each pipeline stage is independent of the others. This is especially advantageous for branches and hazards. Unlike standard pipelined RISC processors which must flush their pipelines whenever a wrong control path is taken, Monsoon never wastes any of the processing done on a token.

The major innovation of Monsoon is its implementation of the waiting/matching area. The technique used, called Explicit Token Store (ETS), allows waiting/matching to be done in unit time. This is accomplished by activation frames. Activation frames are very similar to stack frames found on conventional computers. The base address is known, and variables are referenced by offsets to that base address. Activation frames are created when a codeblock is executed and contain all necessary matching locations along with their presence state bits. Since each token carries its base frame pointer and fetchs its frame offset with its instruction, it knows exactly where it should look for its partner. Activation frames, in addition to making waiting/matching quick, also reduce the amount of memory required. Instead of storing the entire token, including the tag, only the data needs to be stored (recall that tags are normally used for matching.)

Implementing activation frames requires pointers to the frames within the tags of the tokens. A c represents such a pointer. $C.s$ is often written to represent the instruction/frame pointer combination found in a Monsoon tag.

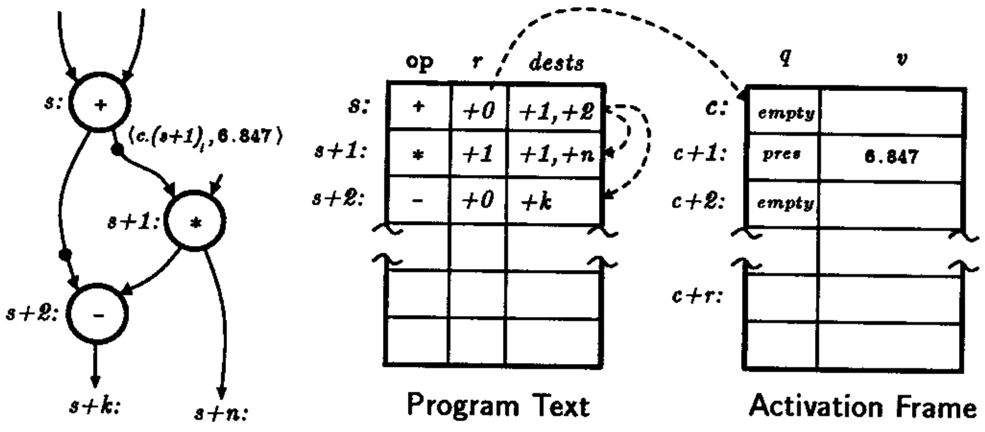
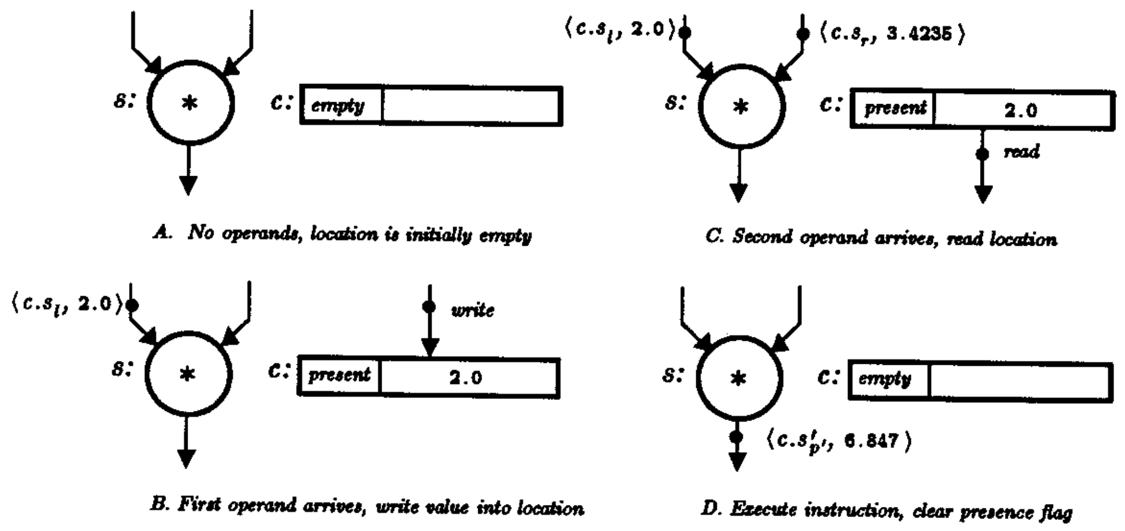


Figure 2.2: Activation Frame and Code

An activation frame and its corresponding program text are shown in Figure 2.2. Notice that the $*$ depends on the $+$, and the $-$ depends on both the $*$ and the $+$.

Monsoon fetches its instruction which tells it the matching location of the token in terms of an offset to the activation frame base pointer, and up to two destinations for its result tokens. An interesting point to notice is that both the + and the − instructions use the same matching location. This is possible because the − instruction can only receive input tokens after the + instruction is done and has reset the presence bits to *empty*.



s:
*
c:
empty
A. No operands, location is initially empty
⟨c.s_l, 2.0⟩
⟨c.s_r, 3.4235⟩
present
2.0
read
C. Second operand arrives, read location
write
B. First operand arrives, write value into location
⟨c.s'_p', 6.847⟩
D. Execute instruction, clear presence flag

Figure 2.3: Token waiting and matching

Another dataflow innovation found in Monsoon is its technique for token matching. Monsoon uses presence bits to indicate the state of a particular matching location in an activation frame. An example of token waiting/matching is shown in Figure 2.3. In the standard tagged-token dataflow model, the check is done on the tags of the tokens which must be stored along with the data. Monsoon checks presence bits associated with the partner's location rather than actually checking for valid data. If the partner is present or the operation is unary, the instruction is fetched and applied to the data (Figure 2.3, part c). Otherwise the data is written to memory where it will wait for its partner (Figure 2.3, part b.) Thus, the basic order of operations is as follows.

- The first token enters its processing element and checks the presence bits of its predetermined meeting point for the state of its partner. Since the state is *empty*, the token knows that its partner has not yet arrived. The token is written to the specified meeting place, and the presence bits of the location are mutated to a *present* state.

- The second token enters the processor some undetermined amount of time afterwards. It checks the presence bits of the predetermined meeting location and finds that the *present* state is set. Thus, it knows that its partner is available, fetches that partner and executes the instruction. The presence bits are reset back to *empty*. This reset is important if the location in the activation frame is to be reused.

This matching technique has a couple of advantages. The first benefit is that the data path and the hardware necessary to check on a match is much less than a system that compares entire tags. Of course, knowing where to look for the partner (ETS) is crucial to this advantage. The second benefit is that a stored token can have more than two states of presence. This is especially advantageous for complicated instructions, such as gates with two triggers.

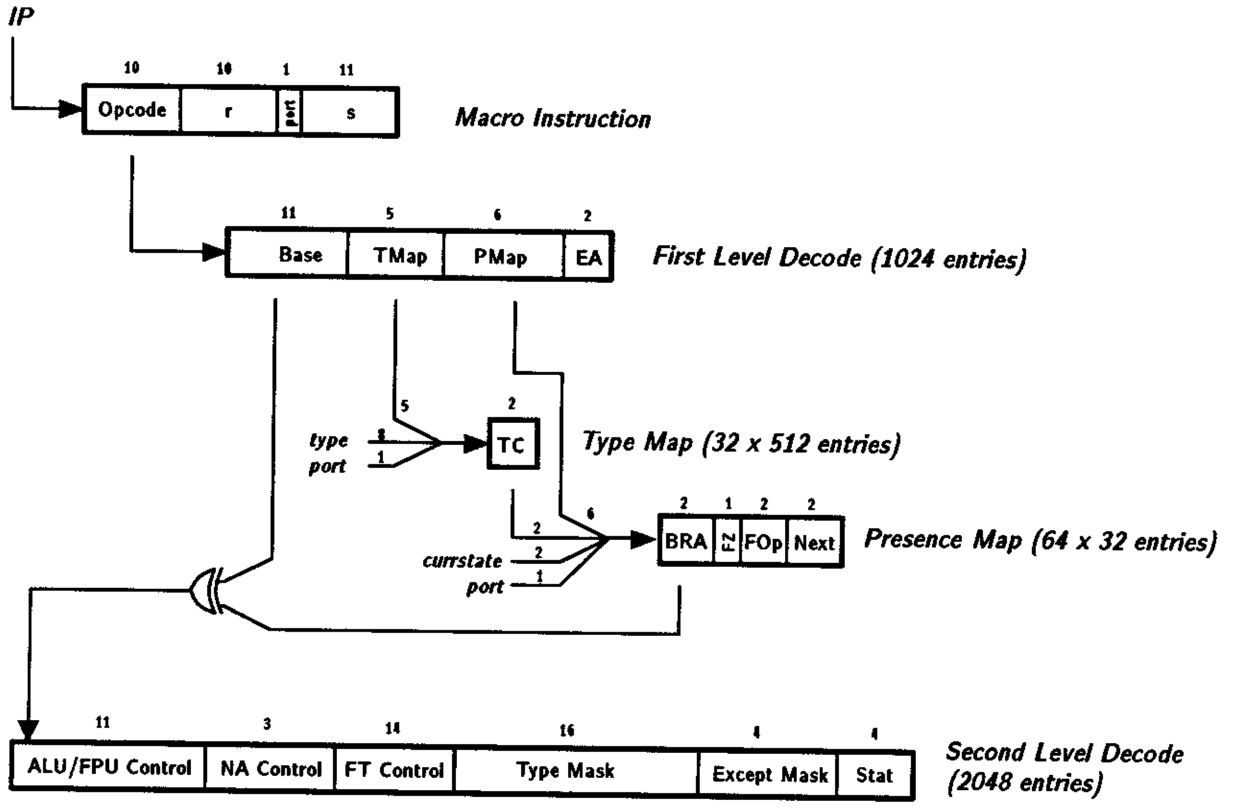


Figure 2.4: Instruction Decoding Tables and Maps

For experimental purposes, Monsoon's pipeline is controlled by a microcode (refer to Figure 2.4.) This microcode is not microcode in the traditional sense of instructions for a microengine processor. Monsoon's microcode controls the actions of each of the pipeline stages. For example, the microcode controls how a token is to be constructed, which functional unit is to execute the instruction, and what kind of memory operations are to be performed. The specific microcode that will be run on a specific token is decided by the instruction pointer inside of the token, the type of the value of the token, the port of the token, and the presence bits of the ETS memory associated with that token.

The decoding process starts with a token coming into the processor. The token consists of a number of fields of data as shown below.

Token			
Tag-part		Value-part	
TYPE	TAG	TYPE	VALUE
8	64	8	64

The structure of the tag is shown below.

TAG				
PORT	MAP	IP	PE	FP
1	7	24	10	22

where,

PORT	Indicates whether the token is coming on the left port (*l*) or the right port (*r*) of the instruction specified by PE:IP. This field is called INPUT-PORT throughout the rest of the document.
MAP	Alias and interleave control. Increments to FP affect PE as specified by MAP. (This field is presently not operational.)
IP	Instruction pointer. The absolute address of an instruction on processor number PE.
PE	Processing element number. For machines with less than 1024 physical processors, the LSBs of PE can be concatenated with the MSBs of FP, extending the physical address space of each PE
FP	Frame pointer. The absolute address of a 72 bit location on processor number PE. PE:FP describes a global address, so a machine is limited to a maximum of 4000 megawords of physical memory.

Once the token enters a processor, its instruction is fetched by reference to its IP (refer again to Figure 2.4). The instruction consists of a OPCODE, offsets r and s and the OUTPUT-PORT. The OPCODE references to the first level decode (1ST-LEVEL-DECODE) table of the microcode. The 1ST-LEVEL-DECODE is comprised of a base pointer (BASE) to the second level decode tables, a type map (TMAP) pointer, a presence map (PMAP) pointer, and an effective address mode (EA). Once we have the 1ST-LEVEL-DECODE, the proper TMAP is referenced using the TYPE of the token's data and its PORT, producing a two bit type code (TC). The effective address is computed in parallel with the computation of TC and is used to read the presence bits (CURRSTATE) associated with the activation frame matching location. TC, CURRSTATE, and the INPUT-PORT of the token are used to reference the proper PMAP from which a presence map entry (PENT) is obtained. A PENT consists of a offset to the BASE of the second level decode (BRA), a force-to-zero bit (FZ) which forces the BASE to zero if true, a fetch and/or store operation (FOP), and the next state (NEXT) of the effective address location.

The second level decode table is referenced by the BASE, which is forced to zero if FZ if true, added to BRA. The returned result, 2ND-LEVEL-DECODE, contains information to control the functional units, the next address unit, and the form token unit. The structure of the 2ND-LEVEL-DECODE is shown below.

Second Level Decode Entry					
FUCTL	NACTL	FTCTL	TMASK	EMASK	STATS
11	3	12	16	10	4

where,

FUCTL	Selects a function unit and specifies its control.
NACTL	Controls the next address generation.
FTCTL	Specifies the form token mode.
TMASK	Specifies operand type checking and propagation.
EMASK	Specifies the exception mask.
STATS	Specifies an increment for one of 16 instruction mix counters.

2.2.1 Function Units

The function unit is a set of four units – an arithmetic unit, a pointer unit, a type unit, and a machine control unit. Only the first two are presently operational. The incoming token value and the read value, if any, are assigned to processor variables names A and B depending on the flip bit and the INPUT-PORT. These values are then used by the function units.

We describe the FUCTL control of the functional units.

FUCTL		
FLIP	UNIT	OP
1	2	8

where,

FLIP	Specifies the $l,r \rightarrow A, B$ mapping.
UNIT	Selects one of the four function units: FALU, PIU, MCU or TPU.
OP	Function unit opcode. Interpreted by each function unit.

The arithmetic unit uses OP as an instruction. The pointer increment unit uses bits from the OP to control its three operational units (PORT, IP, and FP.)

2.2.2 Next Address Control

The next address is controlled by three bits divided as follows:

NACTL	
NA1	NA2
2	1

where,

NA1	Specifies the *tag*1 IP increment of 0,1,2, or 3. PORT is always set to l.
NA2	Specifies the *tag*2 IP increment of 0 or s. PORT is set to the instruction PORT or r.

2.2.3 Form Token Control

The FTCTL field has the following structure:

FTCTL							
EN1	EN2	K1	K2	ORD	RECIRC	STACK	ACK
2	2	2	2	1	2	1	1

where,

EN1, EN2	Specifies the conditional output predicates for *token*1 and *token*2.
K1, K2	Specifies the assembly of *token*1 and *token*2.
ORD	Specifies the relative priority of *token*1 and *token*2.
RECIRC	Controls the recirculation of the higher priority token.
STACK	Specifies the stack for the lower priority token.
ACK	Controls acknowledgment for network packets.

See Papadopoulos's[6] Appendix for details on the microcode.

This unique decoding permits instruction execution to branch any one of eight ways (BRA is two bits and FZ is one bit) based on the type and port of a token along with the presence bits associated with it. This control is extremely general and permits remarkable flexibility for experimentation. The desire of the group to experiment with this powerful computing engine is the driving force behind my thesis.

Chapter 3

Objectives and Approach

The primary objective was to take a microcode specification and an instruction set specification and compile them into MINT routines[4] that accurately and efficiently emulate the Monsoon pipeline. Efficient implies run times comparable to hand-coded procedures. Secondary considerations were code-size and human-readability. The speed of compilation was not regarded as important, but did become somewhat of an issue later in the development stages.

3.1 Naive Approach

As stated in the Introduction, a Monsoon pipeline simulator could be trivial to write. A simple microcode interpreter would suffice. An interpreter, however, is very slow. Creating Lisp routines using tables that map microcode to Lisp functions would be another possible solution. The quality of the resulting emulation routines would depend strongly on the quality of the Lisp compiler used to compile them, but could never produce optimal code without further processing, unless the mappings were exhaustive. A compilation based solely on simple table lookup would probably create opcodes of at least a page a piece. Each of the eight possible branches (which become thirty two if you include unique next state writes) would have to be listed out, and all possibly used variables, regardless of whether or not they are necessary, would have be bound and calculated. Compilation based on table lookup alone would produce very large and slow code. Since different Lisp compilers have different levels of optimization, and none are good enough to optimize blatantly inefficient code, it was decided that the microcode compiler should try to produce the best possible Lisp code.

3.2 Specific Microcode Compiler Optimizations

Given the goals of the thesis, a desired destination code style was developed. This style includes no mutators and no consing. Duplication of code is permitted to minimize execution paths. Bindings should only be bound in an environment that needs them. Speed always takes precedence over size. Lisp compilers are assumed to be stupid.

Given the "execution speed is everything" mentality, the following optimizations were developed.

1. **In-line substitution.** Variables occurring only once within a specific branch of execution are in-line substituted. This is the most frequent optimization within the compiler, and reduces code size considerably. The compiler also uses in-line substitution to eliminate unused expressions that were generated from the microcode specifications. In-line substitution occurs many times in the code following this paragraph. Note that `read-dm-data` and `(+ ip s)` (IP) are in-line substituted into the returned values. Also notice that since IP will only be calculated once in each leg of the branch, it is substituted into each leg rather than being bound above. `(+ fp r)` (effective address) is another in-line substitution found in this example.

```
(DEFALU (IDENTITY-M1)
  (DECLARE (IGNORE VALUE-TYPE))
  (IF (= INPUT-PORT 0)
      (VALUES :ENQUEUE-HP-QUEUE
              TAG-TYPE OUTPUT-PORT MAP PE (+ IP S) FP
              0 VALUE-VALUE)
      (VALUES :ENQUEUE-HP-QUEUE
              TAG-TYPE OUTPUT-PORT MAP PE (+ IP S) FP
              0 (READ-DM-DATA PE (+ FP R)))))
```

2. **Common sub-expression elimination**. As a side-effect of in-line substitution, any binding used by more than one other binding or statement is bound in an environment accessible to those bindings and/or statements. This will eliminate common sub-expressions within bindings and will hopefully make the code run faster. The compiled code for `Identity-M2` is shown. Notice that since `temp-value-*` is used twice in the else-conditional leg, it is bound in that leg.

```
(DEFALU (IDENTITY-M2)
  (DECLARE (IGNORE VALUE-TYPE))
  (IF (= INPUT-PORT 1)
      (LET* ((TEMP-VALUE-* (READ-DM-DATA PE (+ FP R))))
        (VALUES :ENQUEUE-HP-QUEUE
                TAG-TYPE OUTPUT-PORT MAP PE (+ IP S) FP
                0 TEMP-VALUE-*
                :ENQUEUE-PIPELINE-TAIL
                TAG-TYPE 0 MAP PE (1+ IP) FP
                0 TEMP-VALUE-*))
      (VALUES :ENQUEUE-HP-QUEUE
              TAG-TYPE OUTPUT-PORT MAP PE (+ IP S) FP
              0 VALUE-VALUE
              :ENQUEUE-PIPELINE-TAIL
              TAG-TYPE 0 MAP PE (1+ IP) FP
              0 VALUE-VALUE)))
```

3. **Functional symmetry**. Taking advantage of symmetrical functions reduces code size and makes the code faster by eliminating an unnecessary conditional. An example of what can be done knowing that a function is symmetric is found in the following compiled opcode. Notice again that no unnecessary bindings are created and that no unnecessary conditionals are evaluated; in fact, all bindings and conditonals have been eliminated.

```
(DEFALU (+-C1)
  (DECLARE (IGNORE INPUT-PORT VALUE-TYPE))
  (VALUES :ENQUEUE-HP-QUEUE
          TAG-TYPE OUTPUT-PORT MAP PE (+ IP S) FP
          0 (+ (COERCE-TO-FLOAT VALUE-VALUE)
               (READ-DM-DATA-AS-FLOAT PE (+ FP R)))))
```

4. **Binding inside of conditionals.** This optimization eliminates extra calculations when possible. A hacked version of the opcode `i-putr-l2` (enqueue on conditional (b = 0) was added) follows this paragraph. Notice that `b-tag` is bound inside of the conditional since it is not used outside of the conditional, while `temp-value-*` is bound outside of the conditional since the conditional needs it.

```
(DEFALU (I-PUTR-L2)
  (DECLARE (IGNORE INPUT-PORT VALUE-TYPE))
  (LET* ((TEMP-VALUE-* (READ-DM-DATA PE R)))
    (IF (ZEROP (COERCE-TO-INTEGER TEMP-VALUE-*))
        (LET* ((B-TAG (COERCE-TO-TAG TEMP-VALUE-*)))
          (VALUES :ENQUEUE-HP-QUEUE-ENQUEUE-PIPELINE-TAIL
                  PE TAG-TYPE OUTPUT-PORT MAP (+ IP S) FP
                  0 VALUE-VALUE
                  (TAG-PE B-TAG) (READ-DM-TYPE PE R)
                  (TAG-PORT B-TAG) (TAG-MAP B-TAG) (TAG-IP B-TAG)
                  (TAG-FP B-TAG)
                  0 VALUE-VALUE))
        NIL))))
```

5. **Conditionals inside of bindings.** Conditionals between two similar codeblocks can be inserted into the bindings themselves. This drastically reduces code size of some some instructions. A simpler compiler would simply duplicate the code in the two arms of the `input-port` conditional.

```
(DEFALU (--N2)
  (DECLARE (IGNORE VALUE-TYPE))
  (LET* ((RW-DM-DATA-* (RW-DM-DATA PE (+ FP R) VALUE-VALUE)))
    (IF RW-DM-DATA-*
      (LET* ((Y-FLOAT
        (IF (= INPUT-PORT 0)
            (- (COERCE-TO-FLOAT VALUE-VALUE)
                       (COERCE-TO-FLOAT RW-DM-DATA-*))
                    (- (COERCE-TO-FLOAT RW-DM-DATA-*)
                       (COERCE-TO-FLOAT VALUE-VALUE)))))
        (VALUES :ENQUEUE-HP-QUEUE
                TAG-TYPE OUTPUT-PORT MAP PE (+ IP S) FP
                0 Y-FLOAT
                :ENQUEUE-PIPELINE-TAIL
                TAG-TYPE 0 MAP PE (1+ IP) FP
                0 Y-FLOAT))
        NIL)))
```

6. **Flip check removal.** Since the flip bit is static within the instruction, we reduce the flip check to either a port check or nothing at all. Please refer to the previous optimization for an example of how a flip is removed.

7. **Efficient read-write-presence routines.** These routines are features of MINT and claim to be faster than their components executed singly. Combinations of data memory reads and presence and type writes are provided to eliminate unnecessary address lookups. This affects both the speed and the size of the procedure. Examples of efficient memory operations exist throughout the previous examples. Note the **`rw-dm-data`** in **`.--n1`**. Support for the **`write-dm-dtp`** and **`write-presence-and-read-dm-data-as-*`** is also provided.

8. **Elimination of unnecessary reads and writes.** The compiler will optimize all unnecessary reads and writes out of the code. This includes not only reads but write presences as well since a lot of opcodes write the present presence state back as the next presence state. Everything that can be asserted to be unnecessary is eliminated. This optimization tends to be a big win given that memory references are expensive.

9. **Binary decision trees.** This is both a size and a speed issue, though trade-offs between the two may be incurred. If every path is listed separately, code size would grow, since common codeblocks would have to be listed twice (though pointers could be set to common codeblocks, but this would limit readability.) Generated predicates, however, are sometimes larger than ones found in a case statement. It is believed, however, that predicate evaluation is cheaper than chasing pointers. Also, the binary branching should make any large branching cheaper due to its *O(log n)* growth. An example of a rather large operation that takes advantage of the predicate generator follows.

```
(DEFALU (ISTR)
  (DECLARE (IGNORE TAG-TYPE MAP IP OUTPUT-PORT S))
(LET* ((EA-INTEGER (+ FP R)))
  (IF (> (READ-DM-PRESENCE PE EA-INTEGER) 0)
      (IF (= INPUT-PORT 0)
          (LET* ((Y-TAG (COERCE-TO-TAG VALUE-VALUE)))
            (VALUES :ENQUEUE-HP-QUEUE
                    0 (TAG-PORT Y-TAG) (TAG-MAP Y-TAG)
                    (TAG-PE Y-TAG) (TAG-IP Y-TAG) (TAG-FP Y-TAG)
                    (READ-DM-TYPE PE EA-INTEGER)
                    (READ-DM-DATA PE EA-INTEGER)))
          (LET* ((Y-TAG (READ-DM-DATA-AS-TAG PE EA-INTEGER)))
            (MULTIPLE-VALUE-PROG1
              (VALUES :ENQUEUE-HP-QUEUE
                      0 (TAG-PORT Y-TAG) (TAG-MAP Y-TAG)
                      (TAG-PE Y-TAG) (TAG-IP Y-TAG) (TAG-FP Y-TAG)
                      VALUE-TYPE VALUE-VALUE)
              (WRITE-DM-DTP PE EA-INTEGER VALUE-VALUE 1))))
      (IF (= INPUT-PORT 1)
          (WRITE-DM-DTP PE EA-INTEGER VALUE-VALUE 1)
          (WRITE-DM-DTP PE EA-INTEGER VALUE-VALUE 3)))))
```

3.3 Structure and Implementation of Compiler

The Monsoon Microcode Compiler uses output generated by the Monsoon Microcode Assembler (UTOOLS) which is presently being used to generate microcode for the hardware. No modifications to the present system are needed to run the compiler. The microcode information comes in the form of a list of opcodes, lists of presence and type maps, and arrays of first and second level decodes.

The overall structure of the Monsoon Microcode Compiler is shown in Figure 3.1 and is described in the rest of the chapter.

3.3.1 Monsoon-Ucode-Compiler

This is the top level procedure and performs high-level compilation control. The first thing it does is look for duplicate first level decode references. There are many opcodes

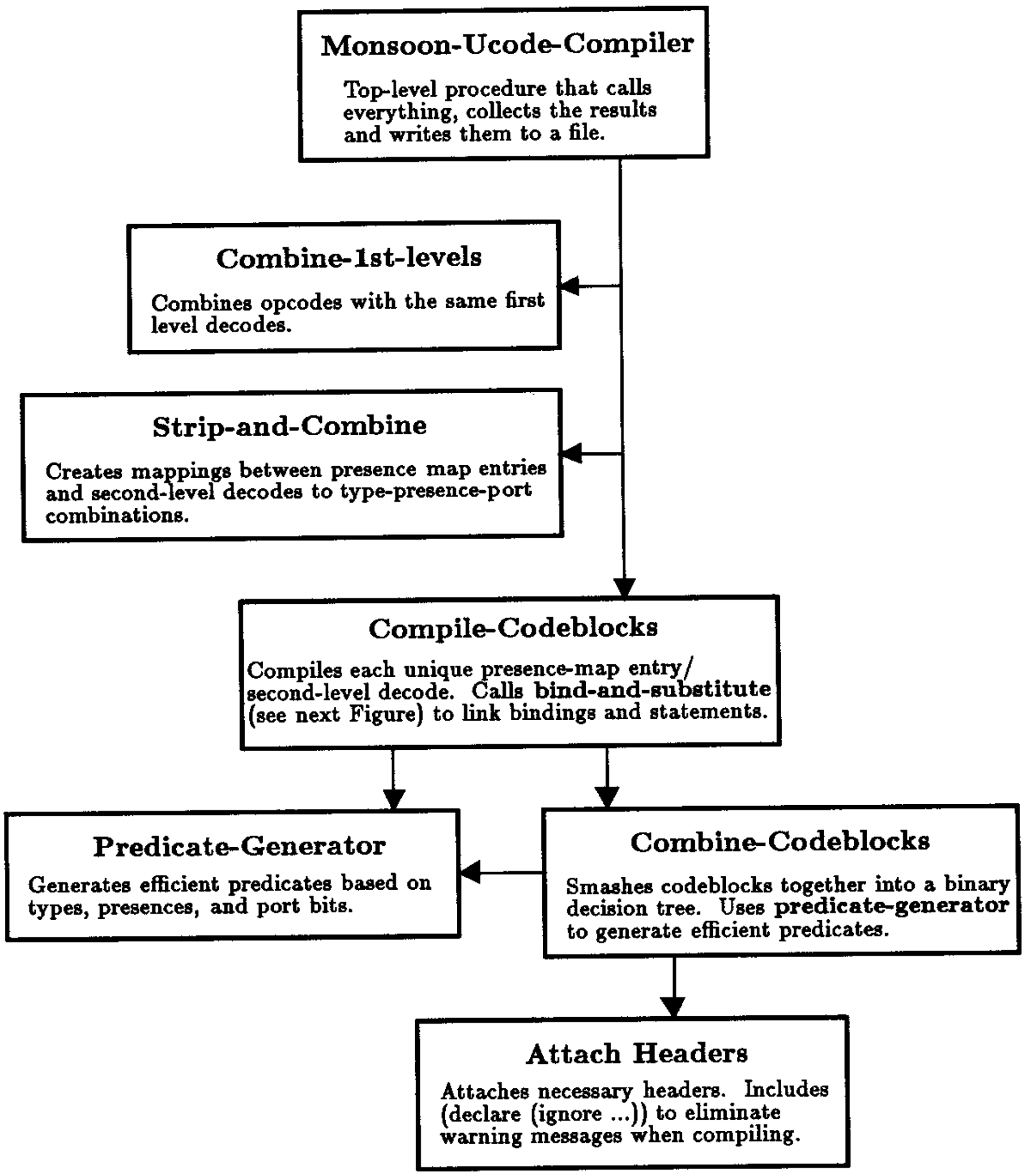

Monsoon-Ucode-Compiler
Top-level procedure that calls everything, collects the results and writes them to a file.
Combine-1st-levels
Combines opcodes with the same first level decodes.
Strip-and-Combine
Creates mappings between presence map entries and second-level decodes to type-presence-port combinations.
Compile-Codeblocks
Compiles each unique presence-map entry/ second-level decode. Calls bind-and-substitute (see next Figure) to link bindings and statements.
Predicate-Generator
Generates efficient predicates based on types, presences, and port bits.
Combine-Codeblocks
Smashes codeblocks together into a binary decision tree. Uses predicate-generator to generate efficient predicates.
Attach Headers
Attaches necessary headers. Includes (declare (ignore ...)) to eliminate warning messages when compiling.

Figure 3.1: Overall structure of the Compiler

that are exactly the same except for their names. The compiler pairs these opcodes up so that work is not duplicated. This is not done if the error option (see Customizing Microcode Tables in the Appendix) is turned on since we need to know the name of the exact instruction we are compiling (error code will generally include the name of the procedure that caused the error.) After the opcodes has been appropriately pre-processed, the compilation starts. Each opcode is processed by the following stages.

3.3.2 Strip-and-Combine

The first level of the actual compiler, **Strip-and-Combine**, compresses the TMAP and PMAP down into a single array referenced by TYPE, INPUT-PORT, and CURRSTATE. Each time one of these smashed TMAP-PMAP combinations is created, it is cached in case another opcode desires the same combination. This compression reduces the complexity of the problem by reducing the number of look-ups necessary and makes the problem a lot easier to understand. Note that duplication occurs throughout the smashed array since eight bits of TYPE and one bit of INPUT-PORT reduce to only four different decodes. An option to turn off the TMAP translation is provided. This makes the compiler faster since it does not have to take care of so many cases.

The compiler then creates unique 2nd-level decodes and presence maps (which we will call UCODE) to presence-port-type mappings. Eliminating duplicates was the first step to compiling as little as possible. Note that each UCODE has a unique combination of PRESENCE, INPUT-PORT, TYPE, and NEXT.

3.3.3 Compile-Codeblocks

This section first checks for special read/write cases that can be optimized using special instructions. If a special case is found, it is processed by alternate procedures. Otherwise, the compiler loops through the list of UCODES produced by **Strip-and-Combine** and compiles them one at a time. The Lisp equivalent of a single UCODE is called a CODEBLOCK. The first step of the compile process is to generate STATEMENTS which are all operations other than bindings performed within a CODEBLOCK. Thus STATEMENTS include token enqueues, writes to memory, statistics writes, and so on. After all of the STATEMENTS are produced, all of the bindings needed by those statements are recursively created. In other words, all bindings which represent all needed values in the CODEBLOCK are produced. These include bindings of every type of every value needed throughout the codeblock. The bindings are generated in order – that is, the bindings are in an order in which they can be computed sequentially.

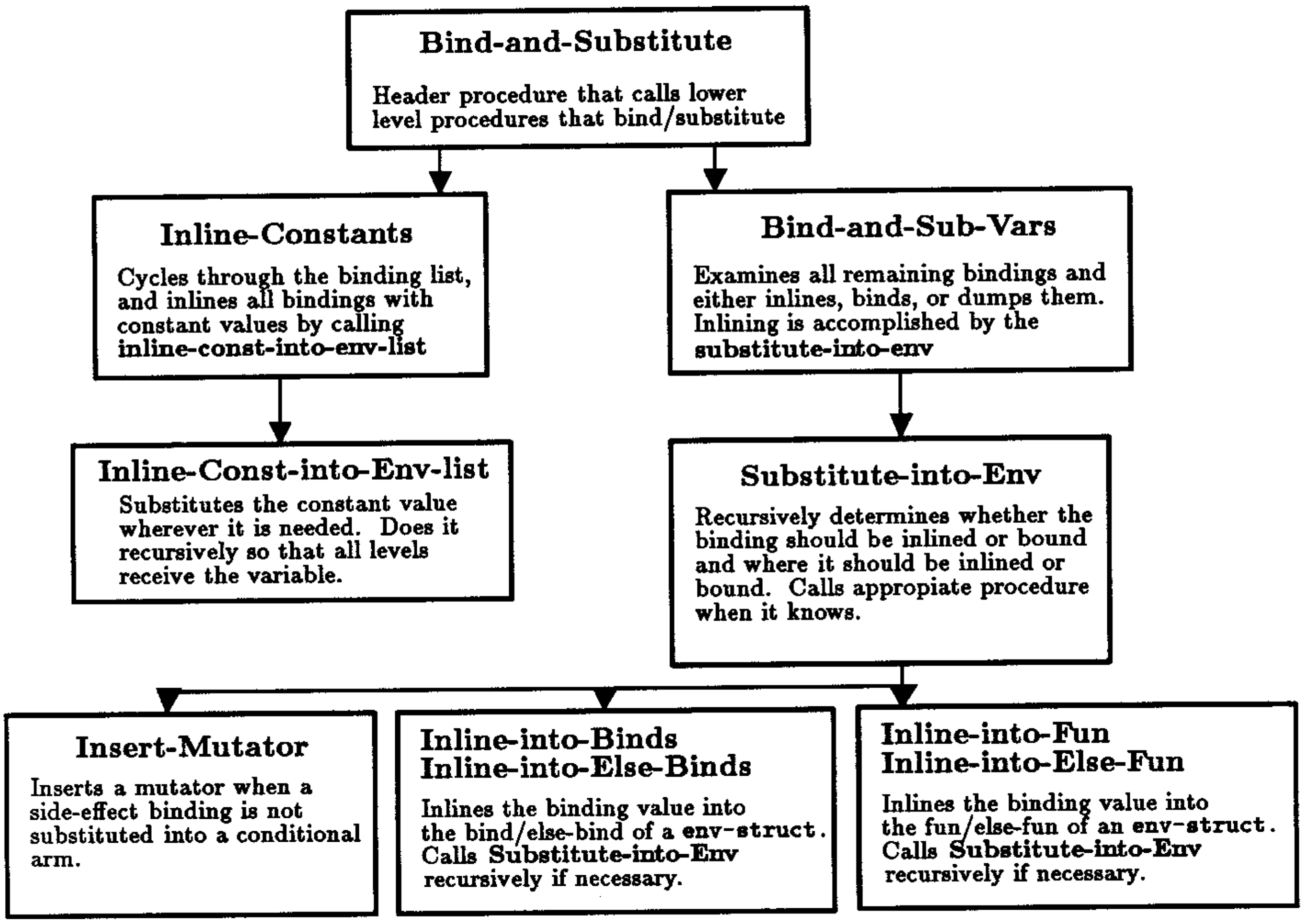

Bind-and-Substitute
Header procedure that calls lower level procedures that bind/substitute
Inline-Constants
Cycles through the binding list, and inlines all bindings with constant values by calling inline-const-into-env-list
Bind-and-Sub-Vars
Examines all remaining bindings and either inlines, binds, or dumps them. Inlining is accomplished by the substitute-into-env
Inline-Const-into-Env-list
Substitutes the constant value wherever it is needed. Does it recursively so that all levels receive the variable.
Substitute-into-Env
Recursively determines whether the binding should be inlined or bound and where it should be inlined or bound. Calls appropiate procedure when it knows.
Insert-Mutator
Inserts a mutator when a side-effect binding is not substituted into a conditional arm.
Inline-into-Binds
Inline-into-Else-Binds
Inlines the binding value into the bind/else-bind of a env-struct. Calls Substitute-into-Env recursively if necessary.
Inline-into-Fun
Inline-into-Else-Fun
Inlines the binding value into the fun/else-fun of an env-struct. Calls Substitute-into-Env recursively if necessary.

Figure 3.2: Bind and Substitute Structure

After the bindings and statements are available, substitution is performed (see Figure 3.2.) Substitution is performed in two waves which are organized into a procedure called **Bind-and-Substitute**. The first wave, called **Inline-Constants** in-line substitutes for all bindings with constant values. The second wave, called **Bind-and-Sub-Vars** counts the number of references to a binding, then binds if the number of references is greater than one, substitutes if the number of references is equal to one, and discards if the number of references is equal zero. This procedure is called recursively to insure that the elimination of a binding is noticed by the bindings above it. **Bind-and-Sub-Vars** calls **Substitute-into-Env** which in turn calls in-line and insertion routines depending on what is appropriate.

The second wave substitution has a number of interesting problems associated with it. One problem involves taking advantage of the typed data memory reads, i.e., reading the data as a specific type. These reads are provided with the hope that they will be more efficient than separate read/coerce instructions. To effectively use the typed-reads, one must know what type is needed. Trouble arises if more than one type of a specific value is needed. The microcode compiler follows a specific algorithm felt to be fairly effective at resolving the typed read problem.

A read binding starts out as a generic read with an unspecified type. If the read binding is in-line substituted into a coercion, the read is mutated into the desired type and the coercion is thrown away. If the read binding is not in-lined into a coercion, it remains an untyped read. Note that if the original read-binding is referenced more than once, it is never in-lined. This implies that a codeblock which uses a read value as two or more different types will bind the read in its untyped form. Binding an untyped read is the correct solution in this case as shown in the following example. Suppose we need an integer then a float. In the case where the original data is a float (which is not unlikely since types are very rarely mixed), unless the original data is saved, two coercions need to be performed. One coercion would be to change the original floating representation into an integer, and the second coercion would be to change the integer representation back into a float. It is worth the extra stack space required to store the uncoerced read to avoid unnecessary coercions.

Another problem associated with substitution is insertion of a binding with a side-effect into a conditional statement. Presently one binding value,
`write-presence-and-read-dm-data` has a side-effect. This instruction is another MINT function designed to be quicker than two separate functions. It writes the presence bits while reading the data memory. If this instruction is substituted into a conditionalized statement, the write must exist in all of the conditional cases. For example, even if the instruction with a side-effect is substituted only into the true arm of the conditional, the false arm should still feel that side-effect. Thus, while substituting the write-presence read-data instruction into a conditional statement, one must be careful to insert a write-presence instruction into any conditional branch not containing the combination instruction. It is better to insert the mutator rather than splitting the combination read/write

instruction because it keeps the advantage of having the combination instruction some of the time.

Once substitution is performed, the codeblock is completed. All other codeblocks that make up the instruction are compiled and accumulated for combination.

3.3.4 Codeblock Combination

Combination can take a number of different paths. The simplest is combining the codeblocks randomly. The codeblocks are split up into a binary conditional tree and predicates are generated to describe the control path through that tree. The predicates would have to be as efficient as possible.

Another scheme, which is the one currently implemented, combines similar codeblocks in an effort to reduce code size. Two codeblocks which differ in the value of one variable (such as the `--N2` instruction: see Optimizations section for the code) can be combined by simply conditionalizing the binding. Dropping the conditional into the binding has the advantage of making the code smaller. Rather than having the conditional on the top and duplicating all of the code, the common code can be shared. This scheme makes the predicate generator necessary, as one cannot know which codeblocks can be combined in this fashion. There is room for improvement in the implemented algorithm. Presently, the conditional is only lifted when the size of the codeblocks are exactly equal, and there is only one difference between the two codeblocks (making the conditional fit over only one statement of the codeblocks.) If the conditionals could be put over more than one statement, this method would become more powerful.

The final codeblock combination scheme is to combine code in a straight conditional fashion – i.e., compare on TYPE, then CURRSTATE, then INPUT-PORT. This will have only one conditional expression at each test point but will contain more test points and more code since duplication is sometimes necessary. I believe, however, that this scheme will be efficient for very small instructions.

3.3.5 Predicate Generator

A predicate generator was built to create efficient predicates around CURRSTATE, INPUT-PORT and TYPE. It is used to support the random combination of codeblocks as described in the previous section. The generator works in a way similar to a Karnaugh map. An array that is referenced by TYPE, INPUT-PORT, and CURRSTATE is marked by *wants*, *don't wants*, and *don't cares*. Solid rectangles are "grown" around the *wants*, including *don't cares* if convenient. The "growing" is done so that the largest rectangles possible are created. Obtaining the largest rectangles involves growing a rectangle, then holding indicies constant and growing again. The largest rectangles that include all wants are then converted into sums of products predicates. Note that both the positive and the negative predicates are generated and only the cheapest is returned and used.

3.3.6 Effective Address and Presence Insertion

Each codeblock knows what its current presence state is and what it's next state should be. Thus, CURRSTATE is never used within a codeblock and, thus, is not in the list of bindings processed during the codeblock binding session. Since CURRSTATE requires EA, the effective address cannot be processed before CURRSTATE. Once combinations are finished and **predicate generation** is complete, however, the effective address and CURRSTATE are dropped in and bound and/or in-lined as necessary. The same routines used in the previous binding processing are used again here. Note that because we are binding CURRSTATE and effective address after we bind everything else, nothing but constants can be used to create these variables. We could wait until the very end to **bind-and-substitute** everything, but it would be very expensive in terms of run-time of the compiler.

After EA and CURRSTATE have been inserted into the instruction, the compilation structures (described in the next section) are removed, the instruction is put into a printable form, and final headers are attached. The compiler prints the completed instruction to the specified file and proceeds to its next instruction.

3.4 Implementation Structures

There are two basic compiler structures that should be discussed. The first of these structures is called `name-struct`. It is defined as a structure with `name`, `minor-name`, and `type` fields. The type field defaults to integer.

The `name-struct` is used in place of a variable. It makes finding variables much easier and gives vital information to the compiler about the type state of that variable. The fields are discussed in detail below.

- **Name** is a Monsoon level name. These include all of the incoming token values such as `output-port`, `fp`, `s`, `r`, `tag-type`, `value-value`, and `value-type`, and the internal names such as `y`, `a`, `b`, `temp-value`, `temp-type`, and `tag-1` and `tag-2`.
- **Minor-names** are presently only used with disassembled tags and indicate the tag part. These include `port`, `fp`, `ip`, etc.
- **Type** is simply the MINT type of the variable. Presently the types include `integer`, `float`, `bits`, `tag`, `*`, and `type`. `*` can stand for any of the first four types.

The second structure used within the microcode compiler is called `env-struct`. This structure is used to represent all statements and bindings within the program. The rationale behind the `env-struct` is that all bindings and statements should have the capability to have environments. Using the same structure for both bindings and statements permits future modifications to be made easily and makes both bindings and statements more flexible.

The structure has as fields `name`, `nom-faux`, `cond`, `cond-vars`, `binds`, `else-binds`, `fun`, `else-fun`, and `needed-vars`. Each of the fields is described in the following itemization.

- **Name** is always a `name-struct` if it is defined. It defaults to `nil`. A `name` exists if the `env-struct` is a bind, otherwise it does not.
- **Nom-faux** is used to indicate a special condition that exists within its `env-struct`. If it is defined, `nom-faux` is always a single `name-struct`. This flag is used mostly for `read-write-presence` bindings so that the **Substitute-into-Env** knows that it is substituting a mutator.
- **Cond** is filled with a conditional statement. If no conditional exists, the default is true.

- **Cond-vars** is a list of variables (`name-structs`) needed by `cond`.
- **Binds** contains a list of bindings that apply if `cond` is evaluated to be true.
- **Else-binds** contains a list of the bindings that apply if `cond` is evaluated to be false.
- **Fun** contains a list of statements which could be `env-structs` and/or Lisp expressions that are executed if the `cond` is evaluated to be true.
- **Else-Fun** contains a list of statements which could be `env-structs` and/or Lisp expressions that are executed if the `cond` is evaluated to be false.
- **Needed-vars** contains a list of variables needed within the conditional. The list includes variables need by all of the bindings and all of the function statements of both the true and false arms of the `env-struct`.

An overview on how `env-struct` is used for bindings and statements follows. Note that brackets <> indicate that that field is optional.

Binding:

```
Name <nom-faux>
  <cond>
    fun
    <else-fun (if cond is not true)>
```

Statement:

```
cond
  bindings
    fun
  <else-bindings> (if cond is not true)
    <else-fun>
```

3.5 Implementation Assumptions

The following assumptions were made during the writing of the compiler. All of the assumptions which had to do with the hardware assume that the hardware is not going to change for specific things. The hardware is not assumed to be totally static but some assumptions made coding easier. It is assumed that the basic functionality of each pipeline stage will not change and their order will not change. Some of the following assumptions clarify basic pipeline assumptions, but most have to deal with the implementation.

- At most one `nom-faux` exists per `env-struct`. This is just to make coding easier. It turns out that more than one `nom-faux` is not presently needed. `Nom-faux's` are only used for tagging bindings with reads and/or writes and for tagging token returns. These are mutually exclusive as a binding is a bind and a token is a statement. Note that using `nom-faux's` to prevent substitutions from crossing boundaries – for instance, substituting a read past a write – might require more than one `nom-faux`. This possibility was not a large one, however; the structure of the pipeline would have to change to require a substitution of a read past a write.

- `Write-presence` has all constant arguments except for `ea`. This is reasonable since the presence bits are written before anything is calculated. This is necessary because a write-presence might be substituted into a token statement before it can be processed by the **bind-and-substitute** routines. This should never happen.

- `Ea` can have only passed variables as parameters. `Presence` can only have `ea` and passed variables as parameters. This assumption is necessary because these two variables are bound and substituted at the very end. It is reasonable because `ea` and `presence` are at the top of the Monsoon pipeline, and no other variables are available.

- All passed variables must use their passed name as their name within their `name-struct`. This makes life a little easier for the compiler.

- If `temp-type` exists, then `temp-value` exists and vice-versa. Take notice of this when writing the `fop-info` array. This makes some matching in **bind-and-substitute** more efficient.

- Passed types are 8 bits big, i.e., they are not larger than 255. This eliminates the need for masking.

- There must be some read and/or write for each `fop`. A fop cannot be a nil. This is not unreasonable since the hardware must also do something for each fop, even if the result is not used. If you are not doing anything for that codeblock, use a read which will be optimized out. This is used by the **check-and-optimize** which assumes if one does not exist, the other does.

Chapter 4

Conclusion

A compiler has been written and is currently generating code that meets all given specifications. The compiled opcodes runs all tested programs at virtually the same speed as hand-coded routines. For example, the benchmark wavefront, which was chosen for its scientific instruction mix, runs on MINT in exactly the same amount of time using either the compiled opcodes or the hand-coded opcodes. Thus, the main objective of the compiler, the speed of its compiled opcodes as compared to hand-coded opcodes, was met.

The compiled output, in addition to meeting speed requirements, matches the original specifications of code style. The code is very readable – the compiler output, as seen in the Optimizations section, is standard. Including types as part of the names of the variables, and not including any Lisp mutators (such as setf, rotatef, etc.) make the code easy to read. The size of the compiled code is comparable to the size of hand-coded code. The size of the binary files of both the compiled and the hand-coded opcodes are comparable as well.

Presently, the compiled opcodes are the default opcodes used by MINT to interpret dataflow graphs. Though the compiler is not quick, taking 770 seconds to compile the entire instruction set on an Explorer II, it is acceptable because of the incremental abilities of the compiler. Since the compiler has the ability to compile a subset of the instruction set, the user is free to compile only what has changed, thus saving time.

It is believed that the techniques used to compile Monsoon's pipeline controls are applicable to RISC processors in general. The routines implemented in this thesis are powerful enough to be used within similar compilers for other processors. Simple alterations to the thesis code could result in microcode compilers for any RISC processor.

4.1 Future Modifications

A few more optimizations could be added. Presently the **combine codeblock** routines are still a bit primitive. Though they catch almost all of the combination optimizations, they do not catch all of them. Those combinations that the compiler does not catch occur very rarely if ever. For this reason, it is believed that improving this section would have minimal advantages.

Predicates could be lifted and bound. Once in a while, some predicates are evaluated more than once because of the way that the codeblocks are combined. Since the **Predicate Generator** prefers equals, which should be extremely cheap, this infrequent redundancy (since most instructions have very few conditionals) should have little performance impact.

An automatic incremental compiler could be added to the system. It would require careful analysis of the entire instruction definition state and a check to see if anything had changed. `Summary.txt` which is written by the Microcode assembler would be of great use here.

Appendix A

Execution

The compiler was designed for ease of compiling an instruction set. When the microcode itself changes, parameters associated with the microcode have to be changed as well. This Appendix attempts to describe way one uses the Monsoon Microcode Compiler.

A.1 Simple Instruction Set Compilation

Most compilations will not require microcode specification changes. To compile an instruction set one needs to load "o:>monsoon>muc>load-muc". Then, call Monsoon-Ucode-Compiler with the correct arguments. This procedure has all of its arguments keyworded and defaulted. The default is compiling the entire instruction set and writing it to
"o:>derek>monsoon-ops.lisp".

The keywords are as follows:

```
(file "o:>derek>monsoon-ops.lisp")
(opcode-list *opcode-list*)
(presence-map-list *presence-map-list*)
(type-map-list *type-map-list*)
(1st-level-decode *1st-level-decode*)
(2nd-level-decode *2nd-level-decode*))
```

Note that `*opcode-list*`, `*presence-map-list*`, `*type-map-list*`, `*1st-level-decode*`, and `*2nd-level-decode*` are all defined by UTOOLS.

A sample run is listed below.

```
>(monsoon-ucode-compiler)

% Then a list of the opcodes that are presently
% being compiled is listed.  This will compile
% the entire instruction set and write it to
% o:>derek>monsoon-ops.lisp
```

If the user wishes to compile a subset of the `*opcode-list*`, he can do so by giving a list in the same format as `*opcode-list*`. This might be useful if only a few opcodes were changed or added. Note that each compile overwrites the old compile if the same file name is used.

A.2 Customizing Microcode Tables

The microcode compiler is customizable to reflect the possibility of changing microcode and internal specifications of the hardware. All specifications[6] are defined as parameters and can be changed easily by the user. Each field of the microcode is defined as an array that is filled with the possible values that that field can take. The array is referenced by the microcode specifications associated with each opcode. An example of such an array is shown below.

```
(defconstant ipop-array (make-array '(4) :initial-contents
                                      `((+ ,*a-ip* ,*b*)
                                        (+ ,*a-ip* ,*s*)
                                        ,*a-ip*
                                        ,*b*)))
```

This array controls the generation of the ipop field of the pointer increment functional unit. Notice that the arguments that need to be substituted are defined as global parameters. These global parameters are simply name structures that make substitution easier. Any variable that needs to be substituted needs to be a name structure.

The opcodes are defined as follows. The *alu-op-list* defines the ALU opcodes. The format of the list is a list of lists of the name of the opcode, the procedure that defines the opcode (in a list), the type of the returned value, and a Boolean that indicates whether or not the opcode is symmetric.

An example of an ALU definition follows. Note that this example is a single element of a list.

```
(FMUL
 (* ,*a-float* ,*b-float*)
 float
 T)
```

The procedure to define opcode can be anything as long as your variable names are correct. The compiler uses straight substitution.

The pointer increment unit operation is defined within arrays that are referenced and copied. Again, these are mutable for customability.

In addition to the functional unit specifications, all other tables as defined by Papadopoulos[6] are defined, some in unusual formats. Stick to the format given in each of the existing arrays. None of the array names can be changed. Unusual formats are enumerated below:

- `k1-array` and `k2-array`. These use a two dimensional array that contain solely symbols that tell the compiler what to use as a tag and as a value in a token. Note that tag-1 and tag-2 must be specified as `'tag-1` and `'tag-2`.
- `tag-1-array` and `tag-2-array`. These arrays contain lists of all of the tag bindings. Make sure to use `make-env-struct-with-needed-vars` in the proper format.