National Library of Canada
Canadian Theses Division
Ottawa, Canada
K1A 0N4

Bibliothèque nationale du Canada
Division des thèses canadiennes

PERMISSION TO MICROFILM — AUTORISATION DE MICROFILMER

- Please print or type — Écrire en lettres moulées ou dactylographier

Full Name of Author — Nom complet de l'auteur
Alan Shelton Wagner

Date of Birth — Date de naissance
02/08/55

Country of Birth — Lieu de naissance
CANADA

Permanent Address — Résidence fixe
C/O A S Wagner
Jordan Falls
Shelburne Co Nova Scotia

Title of Thesis — Titre de la thèse
Verification of S*(QM-1) Microprograms

University — Université
University of Alberta

Degree for which thesis was presented — Grade pour lequel cette thèse fut présentée
M. Sc

Year this degree conferred — Année d'obtention de ce grade
83

Name of Supervisor — Nom du directeur de thèse
Dr. John Tartar

Permission is hereby granted to the NATIONAL LIBRARY OF CANADA to microfilm this thesis and to lend or sell copies of the film.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

L'autorisation est, par la présente, accordée à la BIBLIOTHÈQUE NATIONALE DU CANADA de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans l'autorisation écrite de l'auteur.

Date
July 4, 1983

Signature
A. Wagner

THE UNIVERSITY OF ALBERTA

Verification of S*(QM-1) Microprograms

by

© Alan Shelton Wagner

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH
IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE
OF Master of Science

Department of Computing Science

EDMONTON, ALBERTA
Fall 1983

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR Alan Shelton Wagner

TITLE OF THESIS Verification of S*(QM-1) Microprograms

DEGREE FOR WHICH THESIS WAS PRESENTED Master of Science

YEAR THIS DEGREE GRANTED Fall 1983

Permission is hereby granted to THE UNIVERSITY OF ALBERTA LIBRARY to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

(SIGNED) ..Alan..Wagner.....

PERMANENT ADDRESS:

....JORDAN..FAllS........
....Shelburne.County..
.....NOVA..Scotia........

DATED .June.29.......1983

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled Verification of S*(QM-1) Microprograms submitted by Alan Shelton Wagner in partial fulfilment of the requirements for the degree of Master of Science.

....

Supervisor

....

....

....

....

Date..June 29, 1983.......

To my parents

and

my wife, Jane

Abstract

In recent years, much effort has been devoted to the design and implementation of high level microprogramming languages. One of the goals for such languages is to facilitate the formal verification of microprograms using Hoare's inductive assertion method. Essential to the use of this method is an axiomatic definition of the microprogramming language.

In this thesis, the axiomatization of the machine dependent microprogramming language S*(QM-1) is described. This language is an instantiation of the machine independent language schema S* for the Nanodata QM-1 "nanolevel" architecture. It will be shown that, in spite of the complexity of the QM-1, with its variety of side-effects and special conditions, a pleasingly small and uniform set of proof rules can be constructed.

Acknowledgements

I wish to thank Dr. Subrata Dasgupta for his help and enthusiasm throughout the course of this work. Our many discussions, along with his suggestions and criticisms has contributed greatly to this research.

I am also grateful to Dr. John Tartar for his support and help in the later stages of this work.

Further thanks are due to the other members of my examining committee, Dr. Lee White, Dr. D. Zissos, and Steve Sutphen for their helpful suggestions and comments. I would also like to thank Darrell Makarenko and Ken Hruday for their comments on an earlier draft of this thesis.

Finally, I am indebted to my wife Jane, without whose love and support this would not have been possible.

Table of Contents

Chapter Page

List of Figures

Figure Page

Chapter 1

Introduction

Microprogramming was introduced in 1951 by M.V. Wilkes as a technique for implementing ~~the control~~ unit of compute systems. Since then, mostly due to the technological advances in hardware and innovations in computer architecture, microprogramming has become widely used as a cost-effective and systematic technique for control unit implementation. Not only has the use of microprogrammming increased but recent years have seen a corresponding growth in the size and complexity of microcode [Pat77a].

In response to these increases, along with a demand for added reliability, much attention has been devoted to the development of tools and techniques for the verification of microcode [Dav80a,IEE81a]. Since firmware constitutes one of the lowest levels in the hierarchical, multilevel structure of computer systems, any errors in microcode could have serious and costly repercussions on the reliability of all higher software levels executed on the machine.

In a sense, microprogramming has blurred the once clear distinction between hardware and software. This correspondence between software and firmware has been exploited. The evolution of tools and techniques for the production of microcode has paralleled similar developments in software with the appearance of micro-assemblers followed by high level microprogramming languages (HLMLs) and their compilers [Bab81a,Das78a,Das80a,Dav80b]. The last fifteen years has seen the introduction of a variety of HLMLs along with automated strategies for optimizing and compacting microcode [Dav81a,Fis81a,Lan80a,Rid81a,Tok81a]. It is quite natural then, in view of these developments, that most of the techniques for the verification of microcode are adaptations of similar techniques used in software verification.

As in software, microprogram correctness can be approached using either formal verification or empirical testing. Historically, the most common and easily applied approach has been empirical testing. But is testing adequate?

The disadvantages of testing are well-documented in research on software testing and similarly apply to firmware [Pat76a,Car87a]. Not only is a large proportion of the design time spent testing but the infeasibility of exhaustive testing restricts it to the detection of errors rather than showing their absence. In microprogramming this is further compounded by the difficulty in tracing faults found in microcode back to the error in the microprogram which caused it. These problems which confront testing have identified the need for verification techniques which can be used in conjunction with HLMLs earlier in the microprogram design process. For these reasons, formal verification has been seen as an attractive, alternate approach to the microprogram correctness problem.

Many of the approaches to formal verification directly draw upon the program proof techniques initially formalized by Floyd [Flo67a] and Hoare [Hoa69a]. The rationale behind proofs of programs is that; given assertions concerning the state of program variables during execution and a suitable deductive system for reasoning about these assertions one can produce a rigorous proof of program correctness. Not only can program proving techniques be applied to existing programs but it has been convincingly argued by many authors [Gri81a,Dij76a] that these techniques can be used to *design* correct programs.

Opponents of this radical approach to the design of programs argue that: a) it's not feasible for large programs, and in general, programs found in the "real world", and b) it requires a deductive system which is unnatural and cumbersome to use. But on-going work in this area has resulted in the development of uniform and simple proof rules for many of the constructs appearing in high level languages, including those for specifying parallelism. Secondly, the simple, algorithmic nature of microprograms makes them ideal candidates for formal verification.

Although many of the language constructs in HLMLs appear similar to those in high level languages their semantics may be very different. Microprograms - even those expressed in high level languages - are inherently machine specific [Das80a]. Consequently the formal deductive system for languages like PASCAL [Hoa73a] are not sufficient, nor in some cases, necessary in the microprogramming domain. The various conditions which arise in the host' micro-architecture *must* be taken into account when one considers microcode verification. For example, the data path structure in typical host machines can cause diverse side-effects to be generated in the execution of the most innocuous micro-operations.

The incorporation of machine-specific information into the constructs of a language has been identified as a major problem in the design of HLMLs [Dav80a,Das80a] so it is not surprising that formally representing the semantics of these constructs is a major challenge in the field of microprogram verification. Although there have been a number of proposals for both formally describing the semantics of HLMLs along with different proof techniques, there are only a few actual experimental results reported in the literature. The most notable of these has been the IBM Microcode Certification System by Carter et al [Car78a] using symbolic simulation and the STRUM system by Patterson [Pat77a] using the inductive-assertion method. In particular, Patterson's work has clearly illustrated the feasibility of using proof rules and axioms in a deductive system for proving microprograms.

---

[1] To avoid any terminological confusion the machine which executes a microprogram shall be referred to as the *host* machine (or architecture) and the architecture which a microprogram emulates the *target* machine (or architecture). Note, however, that in the literature on compilers and portability, the machine *for* which the compiler generates code or *to* which a system is proved is referred to as the *target* machine. Thus, the "host" machine to an emulator writer is the "target" machine to the compiler writer!

The general aim of this thesis is to demonstrate how machine specific information can be incorporated into a uniform, and relatively easy to use, deductive system supporting the design of *correct* microprograms. It is also hoped that this thesis will provide some insight into the design of host machines and microprogramming languages which support verification.

More specifically, this thesis shows the construction and use of a formal deductive system for the HLML, S*(QM-1).[2] Developed is a Hoare Logic based on a formal axiomatic definition of S*(QM-1).

Furthermore, this thesis forms a part of the S*(QM-1) project at the University of Alberta. Related work has included, the *instantiation* of S* to the QM-1 [Kla81a], the compaction of microcode produced from S*(QM-1) [Rid81a], and the specification of a C-oriented architecture, QM-C [Ola82a] in this language. In addition to the broader aims, the goal of this thesis in regards to the S*(QM-1) project was to evaluate the schema S* with respect to verification by investigating the verifiability of S*(QM-1) programs.

The rest of this thesis is organized as follows. In chapter 2 the more recent microcode verification systems and proposals are reviewed. Also, the main differences between these efforts and the work contained in this thesis are outlined. Since the main thrust of this work concerns

[2] This language is an *instantiation* of the machine independent microprogramming *schema* S* for the *nanolevel* architecture of the Nanodata QM-1.

programs written in S*(QM-1), the nature of the microprogramming language schema S* and the idea of instantiation of S* to the QM-1 are recapitulated in chapter 3. Chapter 4 provides a brief overview of the QM-1 architecture.

The main results are considered in chapter 5, where the important issues concerning the axiomatization of S*(QM-1) are discussed. Chapter 6 demonstrates how the axioms and proof rules, developed in the preceeding chapter, can be used in the proof of correctness of two different and non-trival nanoprograms. Finally chapter 7 assesses the work reported here and points out some directions for further work.

The entire formal description of S*(QM-1) is contained in a second document submitted as a technical report [Wag83a]. For the sake of completeness, parts of the report contain material already discussed in the main body of the thesis.

Chapter 2

Review and Background

In this chapter several different strategies for formal verification are considered with brief descriptions of actual or proposed firmware verification systems using these strategies. The different approaches to verification can be classified by the type of formal specification used to describe the semantics of the microprogramming language. The semantics of the language can be described denotationally, operationally, or axiomatically.

- **denotational** - language constructs are described as *semantic functions* over the *domain* of values which can be assumed by data objects in the language.
- **operational** - the semantics are described in terms of the more elementary actions which they invoke upon execution.
- **axiomatic** - axioms and rules of inference are used to describe the *properties* of constructs in the language.

Verification techniques based on each of these three types of specifications are presented in the following paragraphs.

A verification technique proposed by Blinkle and Budkowski [Bli76a] uses a denotational description of the semantics of a language. In this method, a microprogram is divided into modules whose binary input/output relation is defined as a set of *semantic functions*. The microprogram, viewed as a combination and functional composition of these modules, is solved algebraically as a system of *fixed-point* *equations*. One finally obtains an input/output relation for the entire microprogram. This method has been successfully tested on a number of microprograms written for a floating point unit.

A verification system proposed by Dembinski and Budkowski [Dem78a] is based upon the Blinkle-Budkowski method and uses a language called MIDDLE (Microprogramming Design and Description Language) and a subset of it, A-MIDDLE (Algorithmic MIDDLE). The authors of this system suggest that the only difference between firmware and software is that firmware is directly linked to the hardware executing it. Therefore, using MIDDLE and A-MIDDLE, one can in a step-wise manner abstract the purely machine-independent behavior from the microcode. This can then be proved correct by using existing software verification techniques.

MIDDLE consists of low level constructs which specify selection, branching, synchronous, and asynchronous assignment. The declaration section in MIDDLE gives a functional description of the hardware components and allows the declaration of higher level functions as combinations of previously defined functions. In the proof itself a set of well-defined transformations are used to change a MIDDLE microprogram into a purely algorithmic A-MIDDLE program after which the Blinkle-Budkowski method is applied to verify the A-MIDDLE program.

Notice that this method is basically a bottom-up system. Starting with the microcode and description of the host architecture, one abstracts away the machine specific properties of the microprogram. Although the authors claim that this system could also be used in a top-down manner for the design of microcode, it remains to be demonstrated that in would in fact generate *efficient* microcode.

Symbolic Simulation is the most widely used verification technique based on an operational specification of the language. This techniques replaces the input data of a program by symbols denoting fixed but unknown quantities, and then simulates the execution of the program on these symbols. The semantics of the language are defined by the simulator in the symbolic execution of the program. This approach has been used in the IBM Microcode Verification System (MCS) developed by Carter *et al* [Car78a].

The MCS system is based on symbolic simulation and Milner's [Mil71a] technique for proving the equivalence of programs. In this method both the behavior of the intended microprogram and architectural description of the host machine is specified in an APL-like language called LSS. Then the completely automated MCS system executes both the behavioral description of the microcode and the actual microcode on the architectural description. The MCS system simplifies and then compares the execution of the two simulations proving their equivalence.

This system was successfully used to verify microcode for the NASA Standard Spaceborn Computer-2 and uncovered a number of errors. However, Carter, has noted that work is still required on the simplification of expressions arising in the symbolic execution and the proof that these expressions are equivalent. More recent work in this area has been in the development of an ISPS[3] based microprogram simulation system developed by Crocker [Cro80a] at ISI (Information Sciences Institute).. But as yet few results have appeared in the literature.

An interesting result related to simulation and microcode verification has been reported by Oakley [Oak79a] at CMU. He has used symbolic simulation to obtain higher-level non-procedural descriptions from ISPS descriptions of the architecture. It seems plausible that this system could be used to generate non-procedural descriptions of microprograms.

The final approach to verification, and the one used in this thesis, is based upon an axiomatic description of the semantics of the language. The axioms and rules of inference given in the description provide the basis for a deductive system used to construct proofs of correctness of microprograms.

The deductive system, or Hoare Logic, consists of formulas in the predicate calculus along with formulas {P} S {Q} where P, Q are predicates and S is some legal program

[3] ISPS is a procedural architecture description language derived from ISP [Bel71a] by Barbacci [Bar82a].

statement or statements. The formula {P} S {Q} is to be read as: if the state of the machine is such that assertion P is true before execution of S then the execution of S leads to a state such that Q is true when (and only if) S terminates. This is a statement of *partial correctness*. A proof of total correctness requires, in addition, a proof that S terminates. P and Q are often called the pre-condition and post-condition, respectively of the statement S.

In addition, a Hoare Logic consists of inference rules from the predicate calculus along with rules of inference which describe the effects of the execution of composite statements in the language. These inferences (proof rules) are usually denoted as

$$\frac{H_1,\ H_2,\ \ldots,\ H_n}{H}$$

which states that whenever the premises H_1, H_2, ... H_n are true then H is also true. The alternative notation (H_1 & H_2 & ... & $H_n \Rightarrow H$) may also be used to mean to same thing. Predicates (or assertions) consist of formulas with a meaningful interpretation with respect to the state of the underlying abstract machine. For instance the assertion language may be founded on the mathematical system of finite binary arithmetic.

Given the pre- and post-conditions, P and Q, one can, in a step-wise fashion, construct valid sequences of statements S for which {P} S {Q} can be shown to be true. An obvious limitation of this method is that the proof is only as good as the pre- and post-conditions. That is, a *proof*, {P} M {Q} of a microprogram M is simply a statement that M is consistent with respect to its pre- and post-conditions. Furthermore, an axiomatization of the language must describe the change of state (of all declared variables in the program) caused by S upon compiling and executing S on the host machine.

Once given an axiomatic definition of the language there are a number of different ways to proceed with the actual proof. The most common approach is the *inductive-assertion* method formulated originally by Floyd [Flo67a] and Hoare [Hoa72a]. In this technique, assertions describing the desired state of the program at particular points in its execution are included in the program. A program is said to be verified, with respect to the assertions, if it can be shown that for every path between these assertions the initial assertion together with the program imply the final assertion. The implications between these two assertions are called *verification conditions* or VC's.

A verification system using the inductive-assertion technique is STRUM developed by Patterson at UCLA. STRUM is a high level microprogramming language oriented primarily to the Burroughs Interpreter (the D machine). In verifying STRUM programs, intermediate assertions are passed to a Verification Condition Generator which automatically uses the axioms and proof rules for STRUM to prove that the microprogram does indeed satisfy the intermediate assertions. This system was used to formally verify a 1700 line microprogram emulating a Hewlett-Packard HP-2115.

As mentioned in chapter 1 the proof technique used in this thesis is similar to the inductive-assertion method except that it is a constructive method for the design of correct programs rather than a method for proving the correctness of existing microprograms. The difference between the two methods is that the only assertions assumed true are the initial and final assertions for the entire program.

If constructive proof techniques are to be successful it is necessary that the microprogrammer have an elegant and easy to use formal deductive system from which he can can derive proofs of program correctness. The formation of an axiomatic definition of S*(QM-1) is quite similar in nature to the formalization of STRUM, referred to earlier. The most important differences between this work and that of Patterson are as follows.

1. STRUM was oriented towards the Burroughs Interpreter while S*(QM-1) is specifically tailored to a somewhat different architecture, namely the QM-1.
2. S* and therefore, any language instantiated from it,

contains a richer set of data structuring capabilities than STRUM. It also contains a number of constructs used for expressing low-level parallelism appropriate for monophase, polyphase, and multicycle timing schemes. STRUM's support for parallelism is relatively simple.

3. The feature known as *residual control* [Fly71a] present in the QM-1 poses rather unique problems in the axiomatization of S*(QM-1). Since residual control is present in several other machines [Kor75a,Kra80a,Str78a] it is hoped that a solution to this problem will be useful in a wider context than for the QM-1.
4. The highly horizontal nature of the QM-1 raises issues concerning side-effects which must be reflected in the axioms and proof rules. Neither STRUM nor any other verification system has addressed this issue.
5. At a more general level the aim of this thesis was to test the viability of the schema S* with respect to verification.

Chapter 3

An Overview of S* and S*(QM-1)

The design of S* as originally conceived [Das78a,Das80a] was influenced primarily by two forces. On the one hand there was a desire to utilize many of the (then) current principles of programming language design and methodology. On the other hand S* was to be formulated so that it:

1. could be instantiated with minimal effort;
2. would facilitate the design, verification, and understanding of well-structured yet efficient microprograms;
3. would allow the representation of microprograms at varying levels of abstraction; and
4. would permit microprograms to be written, verified and understood without reference to the internal organization of the control unit.

Basically, the schema has the following features:

1. The primitive data types bit and sequence and a set of structured data types including, array, tuple, (which is identical to the Pascal record) and stack.
2. A set of *simple* statements which may be used to represent micro-operations - i.e. the most primitive, indivisible units available to the microprogrammer. The set of simple statements includes:
   a. a generic *assignment*, the syntax and semantics of which is not specified in S* - their valid forms and meanings are assumed to be machine-dependent and are

determined during instantiation;

b. the simple *selection* statement

$\underline{\text{if}}\ C_1 \Rightarrow S_1\ ||\ C_2 \Rightarrow S_2\ ||\ \ldots\ ||\ C_n \Rightarrow S_n\ \underline{\text{fi}}$

where the C_i's denote testable conditions and the S_i's are simple statements (other than selections). Here again, the construct merely provides a template for valid selections - the legal testable conditions C_i's and statements S_i are machine-dependent and are determined during instantiation;

c. the procedure call statement; and

d. the goto.

3. A set of *structured* statements that allow for the composition of larger program entities. Most of these are adaptations of Pascal-like statements and, of course, sequential composition. In addition, there exist the cocycle and stcycle constructs for the parallel composition of statements.

4. The synonym declaration statement which allows the programmer to arbitrarily *rename* previously declared data objects or parts thereof.

As previously noted, the syntax and semantics of the constructs in S* are only partially defined. An instantiation of the schema S* to a particular host machine, "M" specifically tailors the constructs in S* to M. The fully defined language thus derived would contain the machine dependent information necessary for the efficient utilization of the micro-architecture. The instantiation process itself can be divided into three stages.

1. The formation of a *data declaration* section. Such pre-defined declarations serve to bind actual machine locations to data objects in the language.
2. The determination of the exact form and meaning of *assignments* and *expressions*. This includes binding S* operators to particular hardware devices and deciding, with respect to the previously declared data objects, what will constitute a valid assignment or expression.
3. the determination of the type of constructs which will be available for *controlling the flow* of the microprograms. This will depend on the control mechanisms available in the machine and, in the case of selection mechanisms, those machine states that are testable.

Such an instantiation was carried out by Klassen [Kla81a] for the QM-1, resulting in the language S*(QM-1).

The primary objective in the instantiation of S*(QM-1) was, within the framework of S*, to design S*(QM-1) so that it would be able to be compiled into efficient object code. In view of this goal, Klassen's work defines the syntax of S*(QM-1) and operationally describes the semantics of the language in terms of the nanoprimitive operations invoked upon execution. The present work, builds upon this definition of S*(QM-1) by making the language verifiable.

As previously outlined, the formation of an axiomatic definition of the language is essential to applying proof techniques to the verification of S*(QM-1) programs. An axiomatization of the language was formed according to the following two constraints. First, important to the application of a constructive proof technique, is the definition of a simple set of axioms with the fewest possible qualifications. Therefore, wherever feasible, the semantics were embedded into the syntax of the language. N only is purely syntactic verification preferable over proofs of semantic correctness [Sto77a] but applying this principle greatly simplifies the formal semantics.

<u>Example 1</u>.

Consider an ALU operation in a host machine which resets the left and right inputs (ail, air, respectively) to zero. If these side-effects are not explicitly bound to the ALU operation then the ALU operation and its side effects can be expressed by the statement sequence:

```
alu_result := alu_expression;
          ail := 0;
          ail := 0;
```

However, alternately the semantics of the ALU operation and its side-effects could be embedded directly into the syntax by means of the multiple assignment statement:

```
alu_result,ail,air := alu_expression,0,0
```

■

Secondly, the present instantiation of S*(QM-1) defines the nano-architecture, data objects, and actions, that must be visible to the programmer in order to produce efficient object code from S*(QM-1) source. In formalizing the language any modifications required to S*(QM-1) in order to support verification must not detract from the need to produce efficient object code.

Chapter 4

A Synopsis of the QM-1 Architecture

In view of the discussions in the preceding chapters on the design and axiomatization of S*(QM-1) it is important to describe the architecture of the QM-1 and the features of it which had considerable impact on this work. The Nanodata QM-1 is a user-microprogrammable general emulation engine with two rather distinctive features: a *two level control store* and extensive use of *residual control*.

The two level control store consists of a higher level *control store* and a lower level *nanostore*. The micro-instructions in control store may interpret the conventional machine instructions which reside in *main store*. These micro-instructions are 18-bit vertical words and have no capacity for specifying concurrent operations. Micro-instructions are, in turn, interpreted by highly horizontal nano-instructions contained in nanostore. At this lower level one can fully utilize the high degree of parallelism possible between nano-operations.

Since the primary goal of S*(QM-1) was to test as stringently as possible the theoretical ideas underlying instantiation, verification, and code compaction, the QM-1 nanolevel architecture was chosen as the testbed for this work. It is at this level, with the distinctive features of the QM-1 and high degree of parallelism available that the more interesting problems in microprogramming arise.

The QM-1 nanolevel architecture can logically be viewed as consisting of a 6-bit domain and an 18-bit domain with data path widths of 6-bits and 18-bits respectively. A block structured diagram of the major hardware components of the QM-1 nano-architecture visible to the S*(QM-1) programmer is given in Fig. 1. Notice that the two domains intersect at the instruction register and that the thirty-two general purpose local store registers serve as a central location for routing values between many of the locations and devices in the 18-bit domain. The 6-bit domain, besides providing support for the storage and manipulation of 6-bit words, acts also as a *residual control* for the control of many of the transfers and transformations occurring in the 18-bit domain.

The idea of residual control originally proposed by Flynn and Rosin [Fly71a], rests on the observation that in emulating a target architecture control information once "set up" remains relatively invariant for significant periods of time. Thus, instead of holding this information in the microword (or, in case of the QM-1, in the nanoword), it can be placed in special registers which can remain invariant for the execution of several microwords. By reducing the amount of control information that needs to be held in the micro(nano)word its width can be significantly reduced.

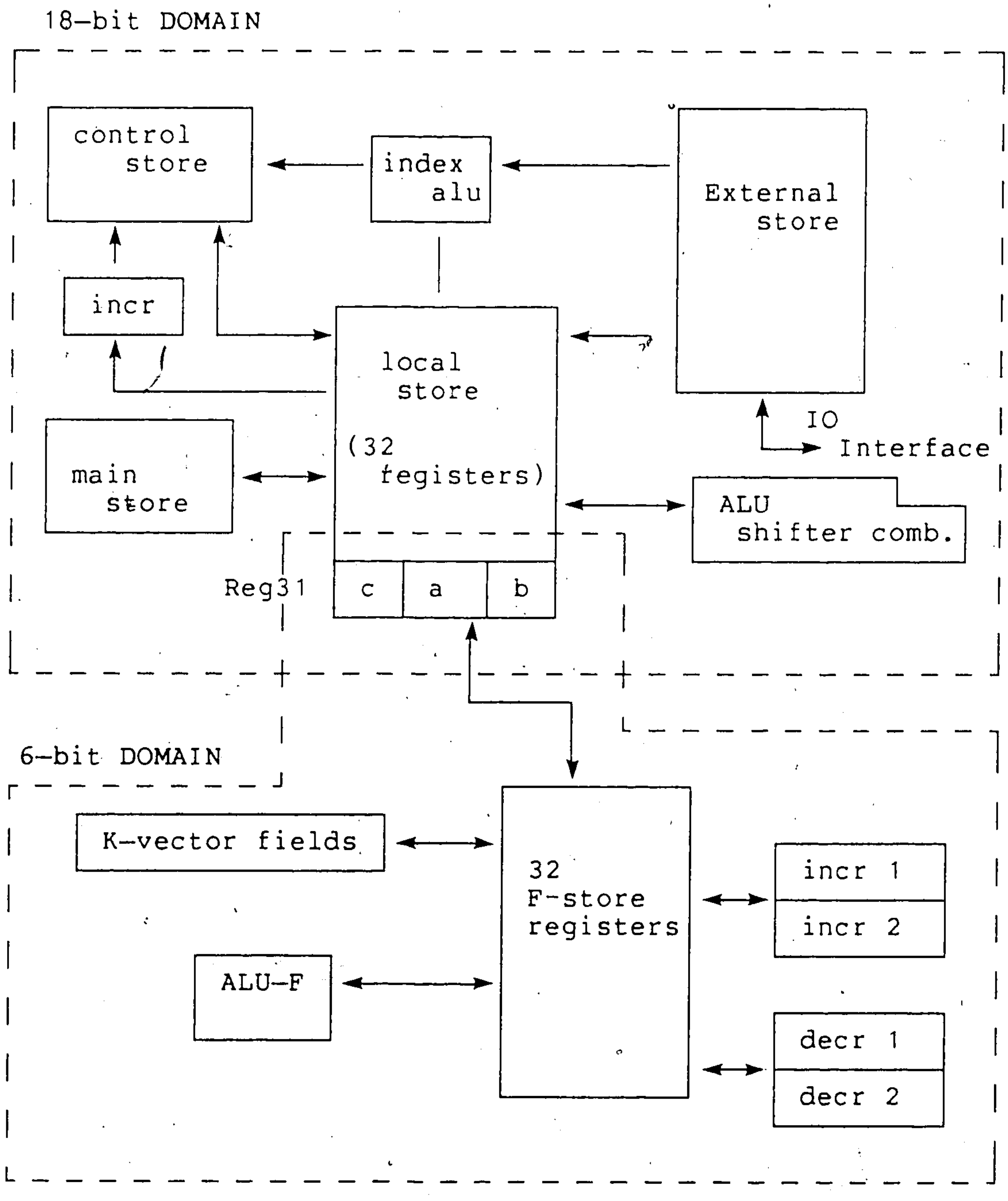
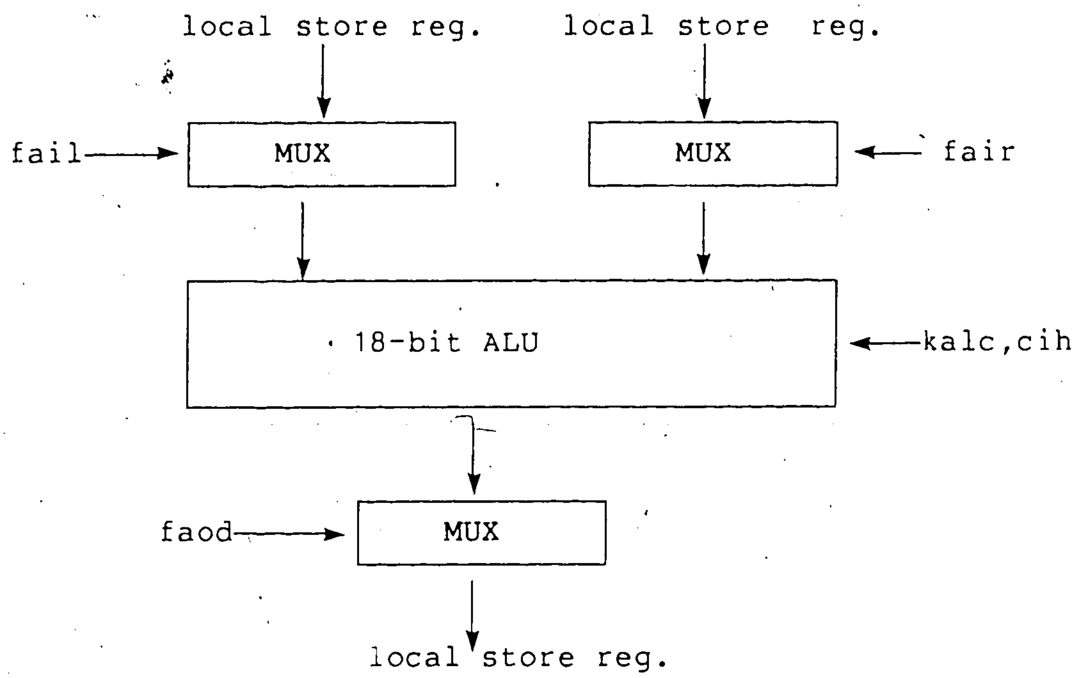


Figure 1. QM-1 Nanolevel Architecture

The most common use of *residual control* in the QM-1 is the selection of local store (or external store) registers for input or output and the selection of the operation to be performed by functional units. For example the F-store registers "fail", and "fair" select the left, and right inputs to the ALU from local store while "faod" determines which local store register is connected to the ALU output bus. The function performed by the ALU is partially determined by the value of yet another residual control register, the K-vector field "kalc". (Fig. 2)

There exists two types of residual control in the QM-1. The F-store registers which remain set until explicitly changed via nanoprogram control and K-vector fields which form part of the executing nanoword. These K-vector fields are considered part of residual control because of the manner in which a nanoword is executed in the QM-1. A nanoword in nanostore is 360 bits wide and is divided into five 72-bit subwords (Fig. 2). On execution of a nanoword the first of these subwords, called the K-vector, remains active throughout the execution of each of the remaining subwords called T-vectors. Each T-vector has an identical format and each are activated in turn, one after another. Thus, from a logical point of view, a combination of the K-vector and the active T-vector constitutes a nano-instruction. Certain of these K-vector fields are not only set or reset on activation of a new nanoword but can also be explicitly changed via nanoprogram control.

In summary, the control function in the QM-1 is quite varied and rests partially in F-store registers which remain stable, K-vector fields remaining stable throughout



K vector	T1 vector	T2 vector	T3 vector	T4 vector

Figure 2. Control Function of Fail, Fair, and Faod and Nanoword Format

the execution of a nanoword, and finally in T-vector fields which provides what Kornerup and Shriver [Kor75a] termed immediate control.

Sequencing between nanowords in the QM-1 is handled by a special priority address mechanism. The address of the next nanoword to be executed could be either of the following: one of 30 interrupt addresses encoded into certain external store registers, a branch address contained in the executing nanoword, or an address supplied by the

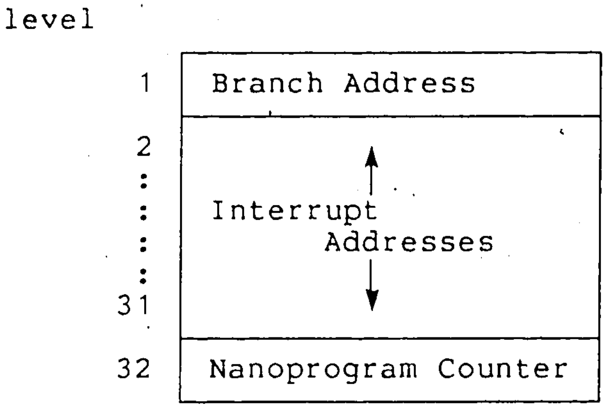


Figure 3. QM-1 Priority Address Mechanism

nanoprogram counter (NPC).

Depending on certain flags set in the executing nanoword each of the addresses in Fig. 3 can become *active*. The priority address mechanism will select the active address with the lowest priority as the address of the next nanoword to be executed. Interrupts become active only if they are *pending*[4] and the allow-interrupts flag has been set in the executing nanoword. The address assigned to the highest priority level, the NPC, is always active and either re-executes the present nanoword, or the next nanoword in sequence.

[4] The interrupt has been enabled and has sensed a 50ns pulse on its signal line.

Chapter 5

Axiomatization of S*(QM-1)

This chapter discusses the key issues which arose in the formation of an axiomatic definition of S*(QM-1). Each of the three stages in the instantiation of S* to the QM-1 along with their effect on the verification of S*(QM-1) code is discussed. Axioms and proof rules are given which describe the semantics of language constructs according to the criteria presented in chapter 3.

The following notation is introduced to describe the semantics of the language constructs in S*(QM-1).

Definition: The logical formula P[x/y] denotes substitution of y for all free occurences of x in P. This corresponds to the more usual Hoare notation where x is the superscript of P and y the subscript of P.

Definition: The formula $P[x_1/y_1][x_2/y_2]...[x_n/y_n]$ denotes the *simultaneous substitution* of all free occurences of the variables x_1, x_2, ... x_n in P by the expressions y_1, y_2,...y_n, respectively. Note that the variables x_1, x_2, ... x_n may only be substituted once, and occurences of some x_i in expressions y_1, y_2,...y_n are not replaced. Furthermore, the substitution is not defined if the variables x_1, x_2, ... x_n are not distinct.

Definition: Substitution formulas containing brackets denote repeated simultaneous substitutions performed on the inner-most bracketed variables first. For example the formula $(P[x_1/y_1])[x_2/y_2]$ denotes substitution of x_1 by y_1 followed by substitution of x_2 by y_2. Note that in this case variables can be substituted more than once. In the previous example if y_1 contains the variable x_2 then y_2 will be substituted for x_2 in y_1.

5.1 Data Declaration

In general, the purpose of the declaration section is to introduce named objects and to designate their properties. Since S*(QM-1) is machine specific, data objects are predefined and correspond to actual locations in the QM-1. The binding of data objects to machine locations greatly affects verification because it defines the state of the machine. The axioms and rules of inference for the language must describe the effects of execution of language constructs upon these data objects.

It was the declaration of the residual control registers in the QM-1 which had the major impact on the axiomatization of the language. Via local store 31 and other sources, these registers can be loaded with values which are not known prior to execution, thereby allowing their control function to be determined dynamically during the execution of the program. Thus, if left undeclared, the programmer is prevented from exploiting the parallelism which is available in the QM-1 by the explicit control of these registers. One effect of the need to declare residual control registers was the introduction of new constructs in the language.

5.1.1 New Types

An important property of structured data types are the selection operations associated with the type. As originally designed in S*, the actual access mechanism of the storage device corresponding to a data object of some structured type is not visible to the programmer. For instance, for an object of type array, elements of the array are refe~~e~~renced by specifying the required element, e.g. local_store[31]. The actual mechanism selecting this location has not been specified. In the case of storage devices in the QM-1 whose elements are selected by residual control registers, because of their visibility to the programmer it is necessary that the associated access mechanism be explicitly described as part of the type. In effect, this is a considerably lower level of description, since it actually specifies part of the underlying data path structure of the QM-1.

The following two types are used in S*(QM-1) for this purpose:

The *array-with-pointer* which was already present in S*, and an entirely new type *union-with-selector*. These types are similar to the array and tuple respectively, except that the selection operation corresponds to the actual access mechanism used in the QM-1 and is completely visible to the programmer.

The general forms of these declarations are as follows:

a) <id$_1$> : array[<dimension>] of <type> with <id>
,<id>}* ↑:<type>↓[5]

where the with ... clause specifies identifiers of the only legal index variables for the array <id$_1$>. That is, the selection of the array element to be accessed is determined solely by values of one of the index variables.

Example 2

In the QM-1 a source of input data for the control store is one of 64 "logical"[6] local store registers, and this would be determined by the setting of the 6-bit residual control register *fcid*. This relationship may be denoted by the declarations

```
    type ls_register = seq [17..0] bit

         .    .    .    .    .    .    .    .

    control_store_data : array [0..63] of ls_register
                                         with fcid
```

Given this declaration the only legal reference to the above structure (say in an assignment statement) would be "control_store_data[fcid]".

■

b) <id$_1$> : selector {<id$_k$>} ↑:<type>↓

[5] The notation {x}* denotes zero or more instances of the entity x while ↑x↓ denotes zero or one instance of entity x.

[6] If the source specified by fcid is greater than 31 the control store data bus is connected to a source of all ones. This is also true for all F-store registers selecting local store registers for output. Therefore, "logically", the set of local store registers consists of 32 registers and a set of 32 non-existent registers whose value is all ones.

```
union
      <id> : <type>
      {<id> : <type>}*
endun
```

Here <id$_k$> acts as a selector of one of the locations enclosed in the union clause. Thus, the union-with-selector type models a hardware multiplexor (Fig. 4). However it is illegal to reference the selector (<id$_k$>) explicitly. Rather, it is *side-effected* upon reference to a location inside the union. If the i-th (i≥0) location is referenced, then <id$_k$> is set to i as a side-effect.

The locations within a union can only be of primitive types bit or seq. Structured types when present are considered to be decomposed into primitive elements.

Example 3

The control store input source of Example 2 can also be declared as:

```
control_store_data  : selector fcid
            union
              local_store  : array[0..31] of ls_register
              all_ones     : array[0..31] of source_all_ones
            endun
```

Given this declaration, a legal reference to this structure is "control_store_data.fcid.local_store[12]". Such a reference would, as a side-effect, set *fcid* to the value 12.

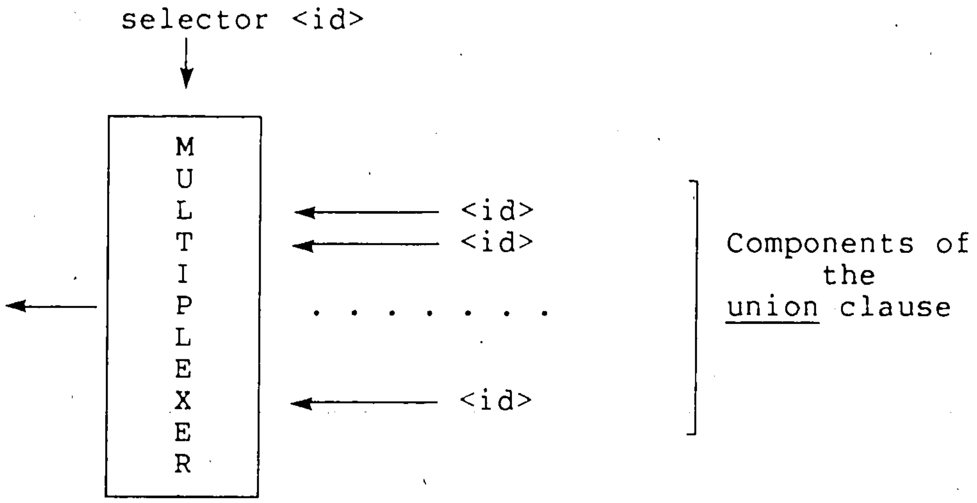


Figure 4. Correspondence of Union-with-Selector Type to QM-1 Multiplexor

In fact, the object nanocode corresponding to this reference would first set *fcid* to 12, and then use this value in *fcid* to access local_store. Note that the reference "control_store_source.fcid.local_store[fcid]" would be illegal.

■

In summary, then, the union-with-selector data type allows the control function of F registers to be specified while the array-with-pointer allows references to storage locations using the preset value of the F register.

5.1.2 "Struct" Declaration

Unlike type declarations in strongly typed languages like PASCAL, S* and S*(QM-1) allow a data object to be of more than one type (i.e. can be a *multitype* object). It is useful in HLMLs to allow different views of the same storage device. For example, in the QM-1 it is convenient to view local store registers not only as an array of registers but also as a structure consisting of varied components. (i.e. instruction register, index register, etc.) The notion of multityped variables was expanded upon by allowing the declaration of new variables as combinations of existing predeclared variables. This was necessary because of the introduction of the array-with-pointer and union-with-selector types. The selection mechanisms in the QM-1 correspond not only to a single storage device but also to different combinations of them.

The part of the declaration which defines new selection mechanisms on predeclared locations is prefixed by the keyword struct (structure). A struct declaration consists only of structured types whose components are either an existing data object or a variable name which itself has been declared in a struct declaration. In the case of multityped declarations at least one declared types must consist of predeclared variable names.

Example 4

```
struct control_store_data
        : array [ 0..63 ] ls_register with fcid
        : selector fcid
          union
            local_store : array [ 0..31 ] of ls_register
            all_ones    : array [ 0..31 ] of source_all_ones
          endun
```

This declaration defines the new access method associated with the type array-with-pointer to the combination of the two predeclared arrays, local_store and all_ones.

■

5.1.3 Pseudovariables

Unlike locations which retain their value until explicitly changed there exist locations in the QM-1 which are unstable and transitory in nature. These locations are declared as pseudovariables and are prefixed by the keyword pvar rather than var in the predefined declaration section of S*(QM-1) programs.

Pseudovariables correspond to two kinds of transient locations in the the QM-1. The first, are the K-vector fields which form part of residual control - since these fields are part of the executing nanoword their values are reset from one nanoword to the next. The second, are locations corresponding to the output buses of functional units in the QM-1 whose inputs are being continuously propagated through the unit. (i.e. ALU, Shifter, Index-ALU) These locations are considered unstable since the input to these devices are local store registers which are selected by residual control registers whose values may not be known prior to execution.

The transient nature of pseudovariables implies that no long term assumptions can be made as to their value. This causes several problems in attempting to prove program segments in which they appear. It can however be guaranteed, by an S*(QM-1) compiler, that once a pseudovariable has been defined it will remain stable at least until the termination of the statement following it - after which it is necessary to assume that its value is undefined. In effect, this couples together the statement defining the state of the pseudovariable with the statement using its value.[7]

Example 5.

```
  pvar alu_out_bus : seq [17..9] bit

      .    .    .    .    .    .

  alu_out_bus := r_alu[fair] + l_alu[fail];
  local_store[faod] := alu_out_bus
```

■

Formally, pseudovariables can be considered to be implicitly side-effected upon execution of an S*(QM-1) statement. Thus, for all pre-conditions P, with assertions containing pseudovariables, the following must hold for the post-condition Q of the statement:

$$Q[\mathrm{PVAR}/\text{undefined value}] \Rightarrow Q \qquad \text{(T1)}$$

where PVAR denotes any pseudovariable appearing in the pre-condition P. The *simple if* statement[8] and all statements inside a cocycle (cf 5.3.1) are exceptions to this rule.

[7] For a discussion of the effect of coupling statements on microcode compaction see [Rid81a,Fis81a].

[8] A *simple if* is an if statement where the body of the if contains only a single assignment, goto, return, or cocycle statement.

Pseudovariables appearing in these statements are not side-effected and remain defined.

5.2 The Axioms of Assignment

In S* the assignment statement is used to model data transfers within the machine. Due mainly to the declarations of residual control registers the assignment statement in S*(QM-1) is not free from side-effects. Thus, it was necessary for an axiomatization of the language to incorporate these side-effects into the definition of S*(QM-1). The manner in which they could be embedded into formal semantics is greatly affected by the decisions made on both, the form of the assignments and expressions in the language and also their correspondence to actual machine operations. The following sections classify the types of side-effects which occur in assignment statements and formally describes their semantics.

5.2.1 Simple Assignment Statement

The simple assignment statement

$$x := y$$

where x,y are locations satisfies the axiom:

$$\{(P[SEL/V])[x/y]\} \qquad (A1)$$

$$x := y$$

$$\{P\}$$

where SEL denotes all selector locations side-effected by reference (if any) to a union-with-selector type, and V denotes the set of values assigned to the selectors. That is, P[SEL/V] is equivalent to $P[sel_1/v_1][sel_2/v_2]$.... The brackets indicate that the side-effects occur before the actual transfer.

Example 6

Let "control_store_data" be declared as in Example 4, and let P be the assertion.

```
{control_store[cs_addr] =
         control_store_data.fcid.local_store[15] & fcid=15}
```

Then by axiom (A1) the following formula is true:

```
{P[fcid/15][control_store[cs_addr]
              /control_store_data.fcid.local_store[15]]}
control_store[cs_addr] :=
                    control_store_data.fcid.local_store[15]
                           {P}
```

On substitution, this reduces to

```
                          {TRUE}
control_store[cs_addr] :=
                    control_store_data.fcid.local_store[15]
                           {P}
```

■

A necessary condition for any transfer to occur is the existence of a direct data path[1] between the source and sink locations of the assignment. For each such statement in an S*(QM-1) program, there must also exist a mapping of the statement to nanoprimitive control fields. If this condition is not satisfied then the statement is not *compilable* and a *mapping error* occurs upon compilation.

5.2.2 Expressions in Assignment Statements

The second class of assignments are of the form

x:=E

where E is an expression.

As noted in chapter 4, a second group of residual control registers - the K vector fields - are part of every nanoword. During the execution of a nanoword each T vector is activated while the K vector remains active throughout. Fields in the K vector specify: operations performed by the 18-bit functional units, mask values for testing, and constants for injection into the 6-bit domain. An expression E in an S*(QM-1) program, containing an operator bound to some functional unit will side-effect the relevant K vector fields by modifying or selecting the function to be performed.

A second source of side-effects are the output lines of devices in the QM-1 declared as pseudovariables. The

[1] A *direct data path* is a path in the QM-1 taking input values to an output location *without* moving data through intermediate locations declared as variables in S*(QM-1). Notice, however, that the value of a pseudovariable (cf. section 5.1.3) may in fact be affected by the transfer.

evaluation of an expression using these devices sets the value of the corresponding pseudovariable to the value of the expression.

Expression evaluation is the major source of side-effects in S*(QM-1).

<u>Example 7</u>

Consider the expression

```
shft_in[fsid] l<<s(5)
```

This will perform a single left logical shift of 5 positions on shft_in[fsid] (i.e. local_store[fsid]) and side effect the K vector fields *kshc* (which encodes the shift function) and *ksha* (which encodes the shift amount). Also the shifter output bus will be set to the value of the expression.

■

<u>Example 8</u>.

S*(QM-1) also allows an operation to be *indirectly* specified. For instance

```
shft_in[fsid] (kshc) (ksha)
```

is again, a shift expression where the type and amount of the shift depends on the values of *kshc* and *ksha*. The only side-effect in this expression will be the setting of the shifter output bus.

■

In comparing Examples 7 and 8 note that the latter expression is at a "lower" level than the former. Thus, there is a trade-off between the level of expressions and the presence or absence of side-effects.

S*(QM-1) also permits *complex* expressions of the form

$$(expr_1)\ expr_2$$

where the expressions are evaluated inside-out. Such expressions correspond to directly connected functional units. In the QM-1, the ALU and the shifter can operate independently or as a single unit performing double shifts on the output of the ALU and the shifter input (Fig. 5).

Example 9.

The expression

(l_alu[fail]+r_alu[fair]) shft_in[fsid] l<<d(5)

specifies a double shift on the ALU output and the local store register pointed to by *fsid*.

■

The axiom for assignments containing expressions is:

$$\{P([SEL/V_1][CNTR/V_2][MOD/V_3][MASK/V_4][OUT/E_i])[x/E]\} \qquad (A2)$$
$$x := E$$
$$\{P\}$$

where CNTR denotes K vector fields which ar^ side-effected as a result of the operations and V_2 denotes the encoded values of these operations. MOD denotes any set of modifier fields side-effected by the operators specified in the expression and V_3 defines the corresponding modifier values appearing in E. MASK denotes the K-field side-effected in the evaluation of the expression if it is a boolean

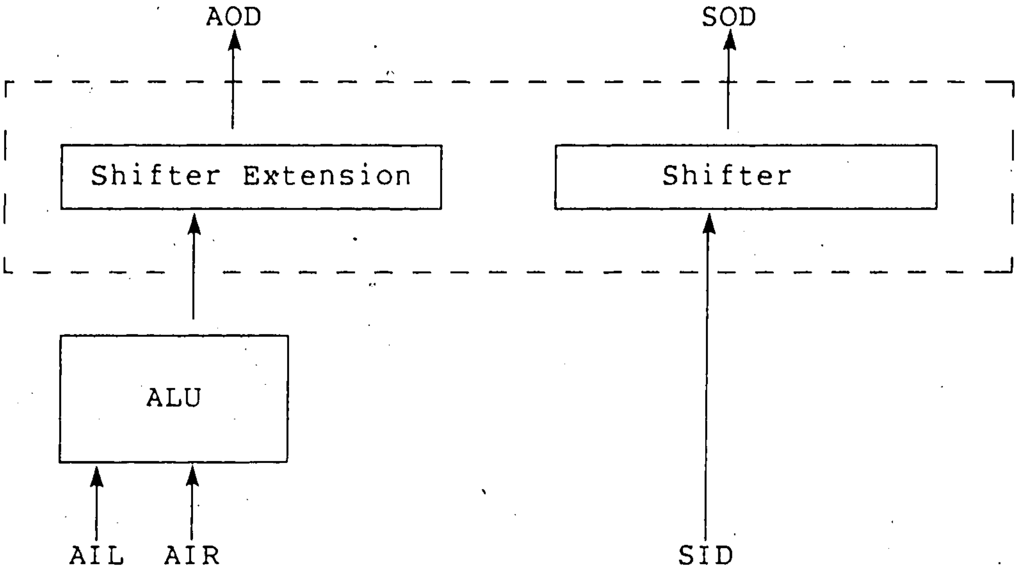


Figure 5. ALU-SHIFTER Combination

expression testing some condition in the QM-1 (cf. section 5.3.2), and V_4 defines its corresponding mask value. Finally OUT denotes any pseudovariables (corresponding to output buses of devices used in E) which are side-effected in the evaluation of the expression and E_i denotes its corresponding expression.

Example 10.

Let P denote the assertion:

{fsod.local_store[15]=fsid.local_store[13] l<<s(5) &
kshc=k_0 & ksha=5 & faod=15 & fsid=13}

where k_0 is a binary (or equivalent integer) valued constant, then, by axiom (A2) the following is true:

{P[faod/15][fsid/13][kshc/k_0]
[shft_out_bus/fsid.local_store[13] l<<s(5)]
[fsod.local_store[15]/fsid.local_store[13] l<<s(5)]}

```
fsod.local_store[15] := fsid.local_store[13] l<<s(5)
                           {P}
```

■

For convenience, the side-effects for expressions $[SEL/V_1][CNTR/V_2][MOD/V_3][MASK/V_4][OUT/E_i]$, will be denoted simply as [EXPR].

5.2.3 Multiple Assignment Statement

There are situations in the QM-1 where the action of functional units result in side-effects on other locations. For example, a write to control store also sets the control store output bus to the value being written. Such actions can be described using the multiple assignment statement which, in S*(QM-1), is of the form:

$$x_1, x_2 := E_1, E_2$$

where x_1, x_2 are variables (or pseudovariables) and E_1, E_2 are valid S*(QM-1) expressions.

<u>Example 11</u>.

```
control_store[cs_addr],cs_output_bus :=
                                  control_source_data[fcid]
```

■

This same statement can be used to specify swapping of values of certain locations in a single time step.[10]

[10] In the QM-1, a *T-step* designates a single step of nanoprogram execution and will, generally speaking, consist of the parallel execution of some set of nano-operations issued from a single T vector. The duration of a T-step is usually 80 nanoseconds (a "T period") although for certain purposes, this may be *stretched* under program control to last for two T periods. All nano-operations are classified as either *leading edge* or *trailing edge* according to whether

e.g. fcid,ka := ka,fcid

The validity of this statement rests on certain transfer-delay characteristics of the QM-1.

The axiom for the multiple assignment statement is:

$$\{P[EXPR_1][EXPR_2][x_1/E_1][x_2/E_2]\} \qquad (A3)$$

$$x_1, x_2 := E_1, E_2$$

$$\{P\}$$

<u>Example 12</u>.

{fsid=2 & ka=3} **fsid,ka := ka,fsid** {ka=2 & fsid=3}

where (ka=2 & fsid=3)[fsid/ka][ka/fsid] ≡ (ka=3 & fsid=2)

■

5.3 Control Constructs

The following sections discuss the effect of instantiation on describing the semantics of control constructs in S*(QM-1). Also included is a discussion on the need for a parallel construct.

5.3.1 Parallelism

An important aspect in the design of S*(QM-1) was to take advantage of the high degree of parallelism available in the QM-1. The major source of parallelism in the 18-bit domain of the QM-1 is the ability to simultaneously gate values in and out of local store (and to a lesser degree

[10](cont'd)the functions they define take effect at the beginning or the end, respectively, of the T step.

external store). Unfortunately, attempting to utilize this ability created a number of problems in the compaction of S*(QM-1) code. In particular, because the input/output to local store is controlled by F-store registers whose values may not be known prior to execution, data interactions (i.e. conflicts) were possible. So as not to seriously degrade the compaction of S*(QM-1) code it was assumed that certain of these interactions would not occur. However, Rideout [Rid81a] does conclude that the introduction of new sequencing constructs could significantly improve compaction. In view of these conclusions and the necessity of ensuring that data interactions do not occur the parallel operator, ¤, is introduced into S*(QM-1).

The parallel composition of two statements

$$S_1 ¤ S_2$$

specifies that the behavior of S_1, S_2 is independent of the order in which they are executed. The statement following $S_1 ¤ S_2$ begins execution only after both S_1 and S_2 terminate.

The basic condition which must be satisfied for the parallel composition of two statements is that they be *interference-free*. Formally, according to the Owicki-Gries [Owi76a] rules for the parallel execution of statements, the *interference-free* condition is defined as follows:

<u>Definition</u>: Statement S', with pre-condition P' is *interference-free* from statement S with pre- and post-conditions P, Q, respectively if:

1. {P' & Q} S' {Q}

2. $\{P' \ \&\ P\}\ S'\ \{P\}$

The proof rule for parallel composition is:

$$\frac{\{P_1\}S_1\{Q_1\},\ \{P_2\}S_2\{Q_2\},\ \text{interference-free}}{\{P_1\ \&\ P_2\}\ S_1\text{¤}S_2\ \{Q_1\ \&\ Q_2\}} \tag{P1}$$

where *interference-free* implies that S_1 is interference free from S_2 and vice-versa.

This is not the only parallelism which is possible in S*(QM-1). Also, instantiated from S*, is the <u>cocycle</u> statement. Unlike the parallel composition of statements the <u>cocycle</u> statement specifies the concurrent execution of all statements appearing in the construct. This concurrent execution is ensured by the requirement that, to be compilable, the control fields initiating the actions specified by the statements in the <u>cocycle</u> must be encoded into a single T-vector and its corresponding K-vector. Formally, since this construct is actually a compiler directive the proof rule is given as:

$$\frac{\{P\}\ S_1\theta S_2\theta\ldots\theta S_n\ \{Q\}}{\{P\}\ \underline{\text{cocycle}}\ S_1\theta S_2\theta\ldots\theta S_n\ \underline{\text{coend}}\ \{Q\}} \tag{P2}$$

where $S_1\theta S_2\theta\ldots\theta S_n$ denotes some sequence of parallel, "¤", and sequential, ";" operators.

5.3.2 Conditional Statements

Like pseudovariables, *testable locations* in the QM-1 are unstable. Unlike the former, however, test conditions are not declared in the data declaration part of the program but are part of the language itself in the form of *test expressions*. Each legal test expression is bound to a particular machine condition.

Example 14.

The machine condition OVERFLOW resulting from an ALU operation is defined by the S*(QM-1) test expression:

```
   LOCAL OVERFLOW of (local_store[fail]+local_store[fair])
```

■

Test conditions in the QM-1 fall into three categories:

1. the "LOCAL" conditions generated from ALU and shift operations - CARRY, SIGN, OVERFLOW, RESULT, SHB, SLB (the latter denoting the high and low order bits of the shifter output bus);
2. These same conditions saved as GLOBAL conditions in a special F register (*fist*); and
3. SPECIAL conditions such as F_REG_ZERO, MS_BUSY and MS_DATA. The latter, for example, is set to 1 if a main store read or write is in progress.

In evaluating a test expression (of the form shown in Example 14, say) additional side effects may occur because of the 6-bit K vector fields *ks*, *kt*, and *kx* which are used as masks for testing the local, global, and special conditions, respectively. The mask for local condition, for instance, is constructed by placing 1's in the bits corresponding to the conditions tested and zero elsewhere. The mask and the test condition are ANDed together with a 1 returned if the result is true, 0 otherwise.

Let "mask_sel" denote one of the keywords LOCAL, GLOBAL, SPECIAL and let "MASK" denote the pseudovariable (i.e. *ks*, *kx*, *kt*) side-effected by the evaluation of the test expression B. Then the proof rules for the if, while, and repeat statements are as follows.

$$\frac{P[MASK/V][EXPR],\ \{P\&B\}S\{Q\},\ P\&\neg B\Rightarrow Q}{\{P\}\ \underline{\text{if}}\ (\text{mask\_sel}\ B\ \underline{\text{of}}\ (E))\ \Rightarrow\ S\ \underline{\text{fi}}\ \{Q\}} \tag{C1}$$

$$\frac{P[MASK/V][EXPR],\ \{P\&B\}S\{P[MASK/V][EXPR]\}}{\{P\}\ \underline{\text{while}}\ \text{mask\_sel}\ B\ \underline{\text{of}}\ (E)\ \underline{\text{do}}\ S\ \underline{\text{od}}\ \{P\&\neg B\}} \tag{C2}$$

$$\frac{\{P\}S\{Q[MASK/V][EXPR]\},\ Q\&\neg B\Rightarrow P}{\{P\}\ \underline{\text{repeat}}\ S\ \underline{\text{until}}\ \text{mask\_sel}\ B\ \underline{\text{of}}\ (E)\ \{Q\&B\}} \tag{C3}$$

5.3. Procedure Statements

There are three types of procedures which may be declared in S*(QM-1): instruction, subroutine, and interrupt. The procedure statements call and act serve to initiate these different procedures and because of the priority select mechanism in the QM-1 their semantics differ significantly from procedure statements in high level languages.

The statement

$$\underline{\text{call}}\ p$$

appears only within an instruction procedure and initiates the subroutine p. The call statement terminates only upon return from the subroutine p. All variables in S*(QM-1) are global so there is no parameter passing but subroutine procedures can be declared with or without an "allow-interrupts" flag. If specified this flag allows the execution of all *pending* interrupt procedures upon return to the calling procedure. Formally, the semantics of this statement are:

1. allow-interrupts option not specified:

$$\frac{\{P\}\ \text{proc}\ p\ \{Q\}}{\{P\}\ \underline{\text{call}}\ p\ \{Q\}} \qquad \text{(P3)}$$

2. allow-interrupt option specified:

$$\frac{\{P\}\ \text{proc}\ p\ \{Q\},\ \forall(i)(\text{int}(i)\Rightarrow(\{Q\}\ \text{proc}\ x_i\ \{Q\}))}{\{P\}\ \underline{\text{call}}\ p\ \{Q\}} \qquad \text{(P4)}$$

The assertion $\forall(i)(int(i)\Rightarrow(\{Q\}\ proc\ x_i\ \{Q\}))$ ensures that the execution of all pending interrupt routines will not affect the post-condition of proc p. The assertion *int(i)* is true only if the interrupt procedure assigned to priority level "i" is now pending.

Example 15.

int(5) is true if the interrupt procedure at priority level 5 is pending and false otherwise.

■

The act statement

act p

activates the named procedure p. It is effectively a goto the start of the procedure. In describing the semantics of this statement the following notation originally introduced by Alagic and Arbib [Ala78a] for describing the semantics of goto statements is used. The notation

$$\{P\}\ proc\ p\ \{Q\}\{p_1:Q_1\}\ldots\{p_n:Q_n\}$$

specifies that Q is true on "normal" exit from procedure p while one of Q_1, Q_2, ... , Q_n is true on exit from procedure p ia activation of p_1, p_2, ... p_n, respectively (Fig. 6).

The semantics of the act statement vary depending on the type of procedure activated. In the case of instruction and subroutine procedures it is:

$$\{P\}\ \underline{act}\ p\ \{\underline{false}\}\{p:P\} \tag{P5}$$

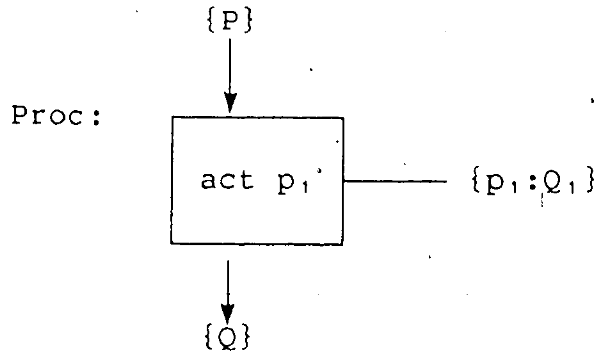


Figure 6. {P} Proc {Q}{p1:Q1} and for interrupt procedures the act statement will not immediately activate the procedure but the interrupt becomes pending.

$$\{P\}\ \underline{act}\ p_i\ \{P\ \&\ int(i)\} \qquad (P6)$$

Again the assertion int(i) indicates that the interrupt procedure assigned to level "i" is now pending (i.e. will be activated at the end of an instruction or call procedure). Notice that the statement act p_i does nothing if p_i is already pending.

5.4 Proof Rules and Axioms for S*(QM-1)

In this section the rules of inference and axioms for S*(QM-1) are summarized.

1. Simple Assignment:

$$\{(P[SEL/V])[x/y]\} \qquad (A1)$$

2. Assignment with Expressions:

$$\{P([SEL/V_1][CNTR/V_2][MOD/V_3][MASK/V_4][OUT/E_i])[x/E]\} \qquad (A2)$$

3. Multiple Assignment:

$$\begin{array}{c}\{P[EXPR_1][EXPR_2][x_1/E_1][x_2/E_2]\} \\ x_1,x_2 := E_1,E_2 \\ \{P\}\end{array} \tag{A3}$$

4. If..fi Statement:

$$\frac{P[MASK/V][EXPR],\ \{P\&B\}S\{Q\},\ P\&\neg B\Rightarrow Q}{\{P\}\ \underline{if}\ (mask\_sel\ B\ \underline{of}\ (E))\ \Rightarrow\ S\ \underline{fi}\ \{Q\}} \tag{C1}$$

5. While do..od Statement:

$$\frac{P[MASK/V][EXPR],\ \{P\&B\}S\{P[MASK/V][EXPR]\}}{\{P\}\ \underline{while}\ mask\_sel\ B\ \underline{of}\ (E)\ \underline{do}\ S\ \underline{od}\ \{P\&\neg B\}} \tag{C2}$$

6. Repeat..until Statement:

$$\frac{\{P\}S\{Q[MASK/V][EXPR]\},\ Q\&\neg B\Rightarrow P}{\{P\}\ \underline{repeat}\ S\ \underline{until}\ mask\_sel\ B\ \underline{of}\ (E)\ \{Q\&B\}} \tag{C3}$$

7. Call Statement(with and without allow-interrupts option specified):

$$\frac{\{P\}\ proc\ p\ \{Q\}}{\{P\}\ \underline{call}\ p\ \{Q\}} \tag{P3}$$

$$\frac{\{P\}\ \text{proc}\ p\ \{Q\},\ \forall(i)(\text{int}(i) \Rightarrow (\{Q\}\ \text{proc}\ x_i\ \{Q\}))}{\{P\}\ \underline{\text{call}}\ p\ \{Q\}} \qquad (P4)$$

8. Act Statement:

$$\{P\}\ \underline{\text{act}}\ p\ \{\underline{\text{false}}\}\{p:P\} \qquad (P5)$$

$$\{P\}\ \underline{\text{act}}\ p_i\ \{P\ \&\ \text{int}(i)\} \qquad (P6)$$

9. Parallel Composition of Statements:

$$\frac{\{P_1\}S_1\{Q_1\},\ \{P_2\}S_2\{Q_2\},\ \text{interference-free}}{\{P_1\ \&\ P_2\}\ S_1\text{¤}S_2\ \{Q_1\ \&\ Q_2\}} \qquad (P1)$$

10. Cocycle...coend Statement:

$$\frac{\{P\}\ S_1\theta S_2\theta\ldots\theta S_n\ \{Q\}}{\{P\}} \qquad (P2)$$

11. Axiom for Pseudovariables:

$$Q[\text{PVAR/undefined value}] \Rightarrow Q \qquad (T1)$$

The following rules of inference are not discussed in this thesis but form part of the axiomatic definition of S*(QM-1). A discussion of the semantics of these constructs is given in [Wag83a].

12. Sequential Composition of Statements:

$$\frac{\{P\}S_1\{Q\}\ \&\ \{Q\}S_2\{R\}}{\{P\}\ S_1;S_2\ \{R\}}$$

13. Case..endcase Statement:

$$\frac{\{P\}\ S_i\ \{Q\}}{\{P\}\ \underline{case}\ ir[17..14]\ \underline{of}\ 0(OP1):S_0...n(OPn):S_n\ \underline{endcase}\ \{Q\ \&\ ir[17..14] = 0\}}$$

14. Goto and Return Statements:

$\{P\}\ \underline{goto}\ L\ \{\underline{false}\}\{L:P\}$ $\qquad$ $\{P\}\ \underline{return}\ \{\underline{false}\}\{E:P\}$

15. Region..endreg Statement:

$$\frac{\{\underline{true}\}}{\{\underline{true}\}\ \underline{region}\ S_1;S_2...;S_n\ \underline{endreg}\ \{Q\}}$$

16. Do..od Statement:

$$\frac{\{P\}\ S_1\theta S_2\theta.\ \ \theta S_n\ \{Q\}}{\{P\}\ \underline{do}\ S_1\theta S_2\theta...\theta S_n\ \underline{od}\ \{Q\}}$$

Chapter 6

Proof of S*(QM-1) Microprograms

In this chapter, two proofs of S*(QM-1) microprograms are presented to illustrate how the axioms and rules of inference developed in the last chapter are to be applied. The two examples chosen are the QM-C multiply (MULT) and call (CALL) instruction. QM-C is the C-oriented architecture developed by Olafsson [Ola82a] for emulation on the QM-1.

These two instructions have been chosen because they both contain loops, and perform two very different functions. The multiply instruction implements a multiplication algorithm while the call instruction must correctly save the contents of certain local store registers. Instructions with loops are presented because the generation of loop invariants usually is the most difficult part in the construction of program proofs. However, as noted by Patterson [Pat76a], loops are in fact quite rare in microprograms. This observation remains true for the S*(QM-1) microprogram written for QM-C, where loops appear only three times. Thus it is hoped, that these proofs will represent the most difficult proofs in the S*(QM-1) code for the QM-C.

In constructing the proof outlines "auxiliary variables" are used to simplify the assertions and enhance their readablity. Such variables appear in assertions but are not contained in the actual code. They are used to denote the value of some machine location at a particular point in the execution of the microprogram. Auxiliary variables will be denoted by subscripting identifiers by "0". (e.g. x_0, abc_0, etc.) Also introduced are capitalized variable names which are used to denote assertions in the microprogram. (e.g. F ≡ {fsid=5 & fsod=3}) These names are used to denote assertions which remain invariant over particular sections of code. Logically, they can be simply viewed as an abbreviation of the actual assertion.

6.1 The MULT Instruction

The QM-C multiply instruction has the format

MULT r1,r2

where r1,r2 specify 18-bit QM-C registers; the effect of this instruction is:

reg[r1] ← reg[r2] * reg[r1]

where "reg" denotes the QM-C register file. The main component of the microroutine interpreting this instruction is a <u>repeat</u> statement. Before entering the <u>repeat</u>, the signs of the multiplier and multiplicand have been determined and both operands converted to unsigned binary numbers.
The instruction register (ir) contains in its two low-order 6-bit fields ir.b and ir.c; the parameters - r1 and r2, respectively of the instruction. The <u>repeat</u> loop is iterated 18 times on the unsigned binary numbers contained in local_store[ir.b] and local_store[ir.c]
Each iteration:

1. computes a partial product, pprod, according to the

expression pprod = pprod + mult[0] * mpcd

2. performs a double-shift-right by one position on the concatenation of sh_end (i.e. the carry of the ALU operation), pprod and mult (Fig. 7)

The entire S*(QM-1) declaration is not given, but is contained in the appendix. However, the following synonym declarations show the mapping of relevant QM-C "logical" registers onto th[illegible] QM-1 data objects.

```
syn fscr1       : r[illegible]31]           /*scratch register*/
syn mult        : l[illegible]store[fsod]   /*multiplier*/
syn pprod       : local_store[faod]         /*partial product*/
syn mpcd        : r_alu[fair]               /*multiplicand*/
```

The S*(QM-1) routine for performing he multiplication is as follows:

```
repeat
   kalc := 9;          /*integer value for ALU "add" operation*/
   if (LOCAL SLB of (mult)) ⇒ kalc := 31 fi;
                   /*integer value for ALU 'pass left' operator*/
   cocycle
       sh_end := CARRY of (pprod (kalc) mpcd) mult d>>l(1);
       pprod,sh_end := (pprod (kalc) mpcd) mult d>>l(1),0;
       mult := (pprod (kalc) mpcd) mult d>>l(1)
   coend;
   fscr1 := fscr1 - 1;
until (SPECIAL F_REG_ZERO of (fscr1)); /*test for fscr1=0*/
```

The pre- and post- conditions for the loop are as follows:

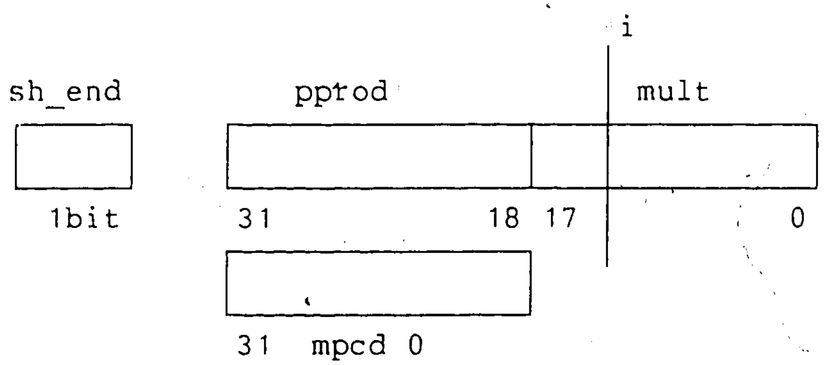


Figure 7. i-th Iteration of Repeat Loop

PRE-CONDITIONS:

1. $answ_o$=mult*mpcd

2. F ≡ {fsid=fsod=ir.b & faod=fail=29 & fair=ir.c}

3. i≡ (18 - fscr1)

4. fscr1=18 & pprod=0

POST-CONDITIONS:

1. F

2. $answ_o$= pprod • mult, where "•" denotes concatenation

Proof Outline

Consider the assertion

INV: { $answ_o$=pprod*2^i+mult[17..18-i]+mult[17-i..0]*mpcd*2^i & F }

On entry into the loop body for the first time, this assertion clearly holds since, initially, by PRE-CONDITION(3,4):

{i = 0}

hence {$answ_o$ = pprod + mult[17..0] * mpcd & F}

and, by PRE-CONDITION(4) pprod=0,

∴ {$answ_o$ = mult * mpcd & F}

which is true by PRE-CONDITION(1,2).

Now, assume that INV holds at the start of the i-th iteration of the loop body. Then for this iteration the following proof outline holds:

$$\{\ answ_0=pprod*2^i+mult[17..18-i]+mult[17-i..0]*mpcd*2^i\ \&\ F\ \}$$

```
    kalc := 9;
      if (LOCAL SLB of (mult)) ⇒ kalc := 31 fi;
```

$$\{\ answ_0=pprod*2^i+mult[17..18-i]+mult[17-i..0]*mpcd*2^i$$
$$\&\ (kalc=9\ \&\ mult[0]=1)\ \vee\ (kalc=31\ \&\ mult[0]=0)\ \&\ F\}$$

```
    cocycle
          sh_end := CARRY of (pprod (kalc) mpcd) mult d>>l(1);
          pprod,sh_end := (pprod (kalc) mpcd) mult d>>l(1),0;
```

$$\{\ answ_0=pprod*2^{i+1}+(pprod+mult[0]*mpcd)[0..0]*2^i$$
$$+mult[17..18-i]+mult[17-i..1]\bullet 0*mpcd*2^i\ \&\ sh\_end=0\ \&\ F\}$$

```
          mult := (pprod (kalc) mpcd) mult d>>l(1)
    coend
```

$$\{\ answ_0=pprod*2^{i+1}+mult[17..18-(i+1)]$$
$$+mult[17-(i+1)..0]*mpcd*2^{i+1}\ \&\ F\ \}$$

```
     fscr1 := fscr1 - 1;
```

{ $answ_0$=pprod*2^i+mult[17..18-i]+mult[17-i..0]*mpcd*2^i & F }
In other words, INV is an *invariant relation* for the loop (i.e. {INV} loop body {INV}).
and

{INV & ¬ SPECIAL F-REG-ZERO <u>of</u> (fscr1) ⇒ INV}

by the proof rule for the <u>repeat</u> statement it can be concluded that

{INV}

<u>repeat</u> ... <u>until</u> (SPECIAL F_REG_ZERO <u>of</u> (fscr1))

{INV & fsrc1=0}

which implies, as the post-condition of the <u>repeat</u> statement:

{ $answ_0$=pprod*2^i+mult[17..18¬i]+mult[17-i..0]*mpcd*2^i & F}
& i=18 & fscr1=0}

By substituting 18 for i in the above and simplifying one obtains the assertion:

{$answ_0$ = pprod • mult & F}

which is the desired POST-CONDITION.

6.2 The CALL Instruction

The objective of the QM-C CALL instruction is to save the contents of the the QM-C registers and to allocate space on the stack before transferring control to the called procedure. The QM-C registers are mapped onto the 32 local store registers, their correspondence is shown in Fig. 8. The following synonym declarations are in effect and relate the QM-C registers shown in Fig. 8 to their corresponding

QM-1 Local Store Registers

		0 : 11
	temporary registers	12 : 15
reg0 : reg7	variable registers	16 : 23
pc	program counter	24
fp	frame pointer	25
eb	external base	26
ax	auxiliary memory pointer	27
sp	stack pointer	28
	scratch registers	29 30
	instruction register	31

Figure 8. QM-C Register File

local store location.

```
syn pc = local_store[24]        /*program counter*/
syn fp = local_store[25]        /*frame pointer*/
syn eb = local_store[26]        /*external base*/
syn ax = local_store[27]        /*auxiliary pointer*/
syn sp = local_store[28]        /*stack pointer*/
```

The stack in QM-C is implemented in control store, its state prior to execution of the CALL instruction is given in Fig. 9.

Saving the contents of the pc, fp and variable registers in the QM-C register file is performed using a repeat loop in S*(QM-1). Before execution of this loop the following is assumed to hold.

1. The instruction register, ir, now contains the value of the "mask-word". (see Fig. 9) The mask-word is the first word of the called procedure and indicates both the lowest number variable register used by the procedure and the space required for storing local variables.
2. The program counter has been incremented for return to the calling procedure.
3. The F-store register "fscr1" now contains the value of the lowest local store register which will be saved. This value is equal to 16+low_reg (as contained in the mask-word).

Given these conditions each iteration of the repeat loop, starting with "fp" (local_store[25]) down to the lowest variable register specified by "fsrc1", saves the contents of the local store register onto the stack in control store. The S*(QM-1) code performing this operation is:

```
repeat
  fcid := fcid - 1;  /*points to register to be saved */
  control_store[cs_addr_source.reg_addr[fcia],cod_bus
```

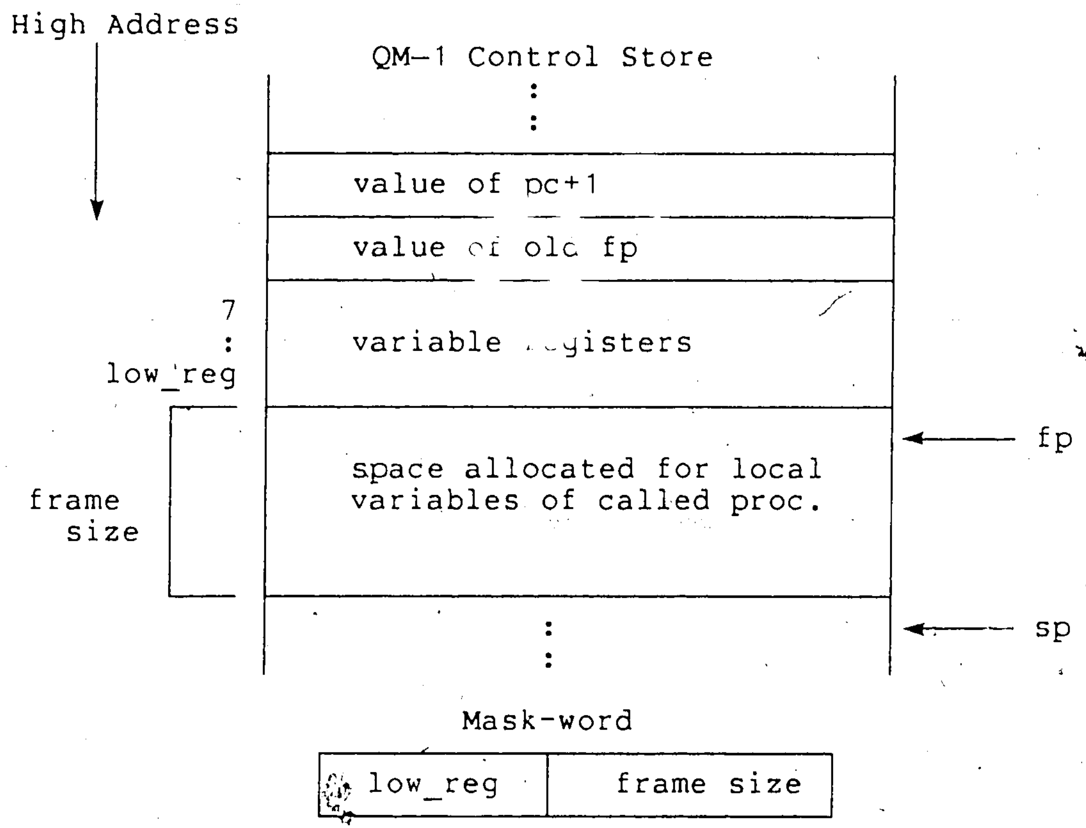


Figure 9. QM-C Stack Frame and Mask-Word

```
                         := control_store_data[fcid];
                  /* saves register on stack in control store */
   kx.local_store[28] := xdecl kx.local_store[28];
                           /* decrements stack pointer */
   fscr2 := fcid - fsrc1;              /* set-up for test */
until  (special F_NOT_ZERO of (fsrc2))
```

Formally, before execution of the CALL instruction the following auxiliary variables are defined.

1. pc_0=pc

2. $sp_0=sp$

3. $\forall(k)(16\le k\le 23)\ reg_0[k-16]=local\_store[k]$

4. $low\_reg_0=control\_store[ir.ab+eb][17..12]\ \&\ 0\le low\_reg_0\le 7$

Now, the pre- and post-conditions of the <u>repeat</u> loop are:

<u>PRE-CONDITIONS:</u>

1. $F\equiv\{\ \forall(k)(16\le k\le 23)\ local\_store[k]=reg_0[k-16]\ \&\ ir=mask$

   $pc=pc_0+1\ \&\ fcia=28\ \&\ fscr1=low\_reg_0+16\ \&\ 17\le fscr1\le 24\}$

2. $\{sp=sp_0\ \&\ fcid=26\}$

<u>POST-CONDITIONS:</u>

1. F

2. $control\_store[sp_0-1]=pc_0+1\ \&\ control\_store[sp_0]=fp_0$

3. $\forall(k)(low\_reg\le k<8)\ control\_store[sp_0-9+k]=reg_0[k]$

4. $sp=sp_0+low\_reg-10$

<u>Proof Outline</u>

Consider the assertion, P:

$\{\forall(k)(fcid\le k<26)\ control\_store[sp_0-k+25]=local\_store[k]$

$\&\ fcid>fscr1\ \&\ sp=sp_0+fcid-26\ \&\ F\}$

On entry into the loop body for the the first time P is clearly true by PRECONDITIONS(1,2). Now, assume that P holds at the start of some iteration of the loop body. Then for this iteration the following proof outline holds.

$\{\ \forall(k)(fcid\le k<26)\ control\_store[sp_0-k+25]=local\_store[k]$

$sp=sp_0+fcid-26\ \&\ fcid>fsrc1\ \&\ F\}$

fcid := fcid -1

{ ∀(k)(fcid<k<26) control_store[sp_0-k+25]=local_store[k]
sp=sp_0+fcid-25 & fcid≥fsrc1 & F}

control_store[cs_addr_source.reg_addr[fcia],cod_bus
:= control_store_data[fcid];

{ ∀(k)(fcid≤k<26) control_store[sp_0-k+25]=local_store[k]
sp=sp_0+fcid-25 & fcid≥fsrc1 & F}

kx.local_store[28] := xdecl kx.local_store[28];

{ ∀(k)(fcid≤k<26) control_store[sp_0-k+25]=local_store[k]
sp=sp_0+fcid-26 & fcid≥fsrc1 & F}

fsrc2 := fcid - fsrc1;

{ ∀(k)(fcid≤k<26) control_store[sp_0-k+25]=local_store[k]
sp=sp_0+fcid-26 & fscr2=fcid-fsrc1 & F}

Now if Q is the assertion given above:

{ ∀(k)(fcid≤k<26) control_store[sp_0-k+25]=local_store[k]
& sp=sp_0+fcid-26 & fscr2=fcid-fsrc1 & F}

then

Q & fsrc2≠0 ⇒ P

Therefore P is the invariant relation for the loop and by the proof rule for the <u>repeat</u> statement it can be concluded that:

$$\{P\}$$

repeat ... until (SPECIAL F_REG_ZERO of (fscr2))

$$\{Q \; \& \; \text{fsrc1}=0\}$$

implying as the post-condition of the loop that:

$$\text{fcid}=\text{fsrc1}=\text{low\_reg}_o+16$$

and

by substituting low_reg_o+16 for fcid and simplifying one obtains the desired post-condition:

$$\begin{aligned}&\{\text{control\_store}[\text{sp}_o]=\text{pc}_o+1 \;\&\; \text{control\_store}[\text{sp}_o-1]=\text{fp}_o\\ &\quad \&\; \forall(k)(\text{low\_reg}\leq k<8)\; \text{control\_store}[\text{sp}_o-9+k]=\text{reg}_o[k]\\ &\qquad\qquad\qquad\qquad \&\; \text{sp}=\text{sp}_o+\text{low\_reg}-10 \;\&\; F\}\end{aligned}$$

Chapter 7

Conclusions

This thesis demonstrates very clearly the issues that arise in microcode verification. It has been shown that although the language constructs are quite similar to those in high level languages their semantics may differ significantly. In constructing a deductive system for S*(QM-1) these differences became apparent in describing: residual control, the different kinds of side-effects, transient locations, and the sequencing mechanisms in the QM-1. Yet, in spite of these differences and with a minimum number of changes to S*(QM-1) it was shown that the axiomatic approach to the verification of S*(QM-1) code was possible. Given that these same issues are characteristic of micro-architectures in general, it is hoped that the form of the axioms and proof rules as well as the relevant extensions to S*(QM-1) necessitated by them, will have wider applications for other host machines and microprogramming languages.

Basically, the following changes to S*(QM-1) were found necessary in order to ensure microprogram verifiability:

1. Side-effects were "classified" and embedded either in the proof rules or in the language itself. Embedding side-effects in the language required the construction of the multiple assignment statement and the novel union-with-selector data type.
2. Testable conditions in the QM-1 were bound to the specific expressions which caused the conditions to

arise. Indeed, unlike the usual assumption made for expressions in programming languages, expressions in S*(QM-1) can yield side-effects.

3. Transient locations were coupled to the statement(s) using their value..
4. The parallel construct, ¤, was introduced for the parallel composition of statements.

A surprising, but pleasant, result of the axiomatization of S*(QM-1) is that, it was possible to produce a reasonably small and uniform set of axioms and proof rules according to the criteria set forth in chapter 3. Even with the complexity of the QM-1 nano-architecture and the variety of side-effects and conditions that may arise, it was still possible to define proof rules with few qualifications. For instance, the proof rule for assignment is valid for *all* assignments in the language encompassing the many different side-effects which can occur.

However, it should be noted that the axiomatization of S*(QM-1) could have been simplified if some of the features of the QM-1 were not present. Although in some cases there is a trade-off between verification and the design of the QM-1 as an efficient host machine there are changes to the architecture which, if done, could simplify verification without compromising its design. These changes are:

1. the removal of those side-effects whose semantics were embedded into the multiple assignment statement. (cf. 5.2.3)

2. the removal of locations which are declared as pseudovariables. For example, the left and right inputs to the alu should be gated into the alu rather than continually propagated through it.
3. the polling of interrupt requests. In the description of the call and act statements it was necessary to restrict the actions of interrupt procedures because of the uncertainty of knowing exactly when the interrupt would be activated.

7.1 Evaluation of S*

Since the constructs in S* are based on the structured constructs in PASCAL, the formation of an axiomatic definition for S*(QM-1) was aided by the semantics presented by Hoare and Wirth for PASCAL [Hoa73a]. The appearance of these constructs in S* allows the application of existing knowledge on the semantics of these constructs to form a basis for analyzing them with respect to languages instantiated from S*.

A second feature of the language which proved useful in describing the semantics of S*(QM-1) was the notion of types. The concept of type in S* is not nearly as powerful as that in high level languages, since in the microprogramming domain compilablility must also be considered. Yet, as demonstrated by its use in both the union-with-selector and array-with-pointer, types can play an important role in providing a lower level description of the underlying architecture. The introduction of these types in S*(QM-1) enabled the control function of many of the residual control registers to be incorporated into the semantics of the language.

It must however be noted that it was not possible to define S*(QM-1) completely within the framework of S*. Recall that one of the aims of S* was the construction of microcode *without* reference to control store organization. Yet it was precisely this control information which had to be incorporated into S*(QM-1). This indicates, at least in the case of very horizontal architectures, it may not be possible to abstract entirely the control structure of the micro-architecture and that in these cases a much lower level of description is required.

As a final point, an indirect consequence of this work is that the very act of formalizing the semantics of the language raised immeasurably, one's understanding of the really *important* properties of the QM-1. In a sense, S*(QM-1) and its semantics is a precise and (hopefully) unambiguous model of the Q nano-architecture which abstracts the variety of behavioral characteristics of the machine into a small set of concepts. The exercise of deriving an axiomatic definition of S*(QM-1) was worthwhile from this viewpoint alone.

7.2 Further Work

A problem of some concern which becomes evident from the proofs presented in chapter 6 is the very low level of description and verification that may have to be carried out in the case of horizontal programs to ensure that the programmer is able to utilize machine resources both efficiently and correctly. Indeed, there is a substantial gap between the larger goal of a microcode routine (e.g. the multiplication of the contents of two register operands) and the final form of the S*(QM-1) program. It is true that the language does permit, to a certain extent, descriptions at different levels of abstraction. An ALU or shift operation can be specified either by a keyword (+,-,and,etc.) or by actually specifying the value of the *kalc* or *kshc* fields, local store operands of the ALU may be specified directly (fair.r_alu[3], fail.l_alu[4]) or indirectly and at a lower abstraction level (l_alu[fail], r_alu[fair]). But, since every construct in S*(QM-1) is bound to the QM-1 hardware so the *range* of the abstraction level is not nearly broad enough.

Dasgupta [Das82b] has suggested that one way of reducing the gap between the broad, abstract, function and the concrete, detailed, form is to use a *family of closely related* (or "kin") languages each designed for, and oriented towards, a specific abstraction level or mode of description. For example a machine independent, operational, architecture description language called S*A [Das82a] together with S* form the first members of an open-ended language family. Also discussed by Dasgupta [Das82b] is how S*A and S*(QM-1) can be used in the systematic design of a target architecture (in S*A) and its QM-1 based emulator, QM-C (in S*(QM-1)). The language S*A was also axiomatized so that it could be formally verified [Das83a].

Experience with the work reported here reinforces the idea that such families of languages are *psychologically necessary* if control is to be maintained over the firmware design process. Additionally, it increases confidence in the correctness of the design while giving a consistent and unified means of documenting the distinct stages of the design. An important question that remains to be answered is: "given an architecture design and it's description in a "higher" level language such as S*A, how is this to be transformed into lower levels of description, such as S*(QM-1), and demonstrably preserve its correctness?"

A second issue requiring closer examination is the exact form of the assertion language used in the proofs of correctness. Again, because of the low-level nature of microprograms, problems are encountered in constructing and manipulating the assertions in the proofs. For example, variables appearing in assertions must be interpreted not only as bit sequences but also as the integer values they represent. The scarcity of theorems on finite bit arithmetic makes simplifying mixed expressions containing arithmetic and logical operators quite difficult. This problem became evident in constructing the proof for the multiplication instruction in chapter 6. The algorithm used both shifting and concatenation operators which had to be interpreted with respect to the more usual 2's complement representation of the bit sequences. More specifically what is required is:

1. A canonical form for representing the value of expressions in assertions. This is certainly needed in view of the several different representations available (i.e. 2'complement, 1'complement, sign and magnitude etc.).
2. A body of knowledge to help in the simplification of mixed expressions and also transformations between the different representations.

A solution to this problem is crucial, if as claimed, a constructive proof technique is to remain an intellectually manageable process. Clearly a complex and unnatural assertion language detracts from the general usefulness of microprogram verification by increasing the likelihood of error in the actual proof.

References

[Ala78a.] Alagic S. and Arbib M.A., *The Design of We Structured Programs*, Springer-Verlag, N.Y., Heideberg, Berlin (1978).

[Bar79a.] Barbacci M.R., "Instruction Set Processor Specifications (ISPS): The Notation and its Applications," Technical Report (CMU-CS-79-123), Carnegie-Mellon University (1979).

[Bab81a.] Baba T. and Hagiwara H., "The MPG System: A Machine Independent Efficient Microprogram Generator," *IEEE Trans. Comput.*, Vol. C-30, pp. 373-395 (June 1981).

[Bel71a.] Bell C.G. and Newell A., *Computer Structures: Readings and Examples*, McGraw Hill (1971).

[Ber80a.] Berg H.K., *Correctness of Firmware -An Overview*, Springer-Verlag, New York (1980).

[Bli76a.] Blikle A. and Budkowski S., "Certification of Microprograms by an Algebraic Method," *Proc. 9th Annual Microprogramming Workshop MICRO 9*, N.Y., pp. 9-14, ACM/IEEE (1976).

[Car78a.] Carter W.C., Joyner W.H., and Brand D., "Microprogram Verification Considered Necessary," *Proc. Natl. Comput. Conf.*, Arlington, VA, pp. 657-664, AFIPS Press (1978).

[Cro80a.] Crocker S.D., Marcus L., and D., Van-Mierop, "The ISI Microcode Verification System," in *Firmware, Microprogramming and Restructurable Hardware*, ed. G. Chroust and J. Mulbacker, North-Holland, Amsterdam

(1980).

[Das78a.] Dasgupta S., "Towards a Microprogramming Language Schema," *Proc. 11th Annual Microprogramming Workshop MICRO 11*, N.Y., pp. 144-153, ACM/IEEE (1978).

[Das80a.] Dasgupta S., "Some Aspects of High-Level Microprogramming," *ACM Computing Surveys*, Vol. 12(3), pp. 295-324, ACM (1980).

[Das82a.] Dasgupta S., "Computer Design and Description Languages," in *Advances in Computers*, ed. M.C. Yovits, Academic Press, N.Y. (1982).

[Das83a.] Dasgupta S., "On the Verification of Computer Architectures Using an Architecture Description Language," *Proc. 10th Annual Int. Symp. on Comput. Architecture*, IEEE Comput. Soc. Press ((forthcoming) 1983).

[Das82b.] Dasgupta S. and Olafsson M., "Towards a Family of Languages for the Design and Implementation of Machine Architectures," *Proc. 9th Annual Symp. on Comput. Architecture*, pp. 158-170, IEEE Comput. Soc. Press (1982).

[Dav80a.] Davidson S. and Shriver B.D., "Firmware Engineering: An Extensive Update," pp. 1-30 in *Firmware, Microprogramming and Restructurable Hardware*, ed. G. Chroust and J. Mulbacker, North-Holland, Amsterdam (1980).

[Dav80b.] Davidson S. and Shriver B.D., "MARBLE: A High Level Machine Independent Language for Microprogramming," pp. 253-263 in *Firmware, Microprogramming and Restructurable Hardware*, ed. G. Chroust and J. Mulbacker, North-Holland, Amsterdam (1980).

[Dav81a.] Davidson S., Landskov D., Shriver B.D., and Mallett P.W., "Some Experiments in Local Microcode Compaction for Horizontal Machines," *IEEE Trans Comput.*, Vol. C-30(7) (1981).

[Bak80a.] de Bakker J., *Mathematical Theory of Program Correctness*, Prentice-Hall, Englewood Cliffs, N.J. (1980).

[Dem78a.] Dembinski P. and Budkowski S., "An Introduction to the Verification Oriented Microprogramming Language MIDDLE," *Proc. 9th Annual Microprogramming Workshop MICRO 11*, N.Y., pp. 139-143, ACM/IEEE (1978).

[DeM79a.] DeMillo R.A., Lipton R.J., and Perlis A.J., "Social Processes and Proofs of Theorems and Programs," *Comm. of the ACM*, Vol. 22(5) (1979).

[Dij76a.] Dijkstra E.W., *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, N.J. (1976).

[Fis81a.] Fisher J.A., "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Trans Comput.*, Vol. C-30(7), pp. 478-490 (1981).

[Fly71a.] Flynn M.J. and Rosin R.F., "Microprogramming: An Introduction and Viewpoint," *IEEE Trans Comput.*, Vol. C-20(7), pp. 727-731 (1971).

[Flo67a.] Floyd R.W., "Assigning Meanings to Programs," *Mathematical Aspects of Computer Science*, Vol. **xix**, pp. 19-32, Amer. Math. Soc. (1967).

[Gri81a.] Gries D.G., *The Science of Programming*, Springer-Verlag, New York (1981).

[Hoa69a.] Hoare C.A.R., "An Axiomatic Basis for Computer Programming," *Comm. of the ACM*, Vol. 12(10), pp. 576-583 (1969).

[Hoa72a.] Hoare C.A.R., "Notes on Data Structuring," in *Structured Programming*, ed. O.J. Dahl, E.W. Dijkstra, C.A.R. Hoare, Academic Press, London and New York (1972).

[Hoa73a.] Hoare C.A.R. and Wirth N., "An Axiomatic Definition of the Programming Language Pascal," *Acta Informatica*, Vol. 2, pp. 335-355 (1973).

[IEE81a.] IEEE, "Special Issue on Microprogramming: Tools and Techniques," *IEEE Trans. Comput.*, Vol. C-30(7) (1981).

[Kla81a.] Klassen A. and Dasgupta S., "S*(QM-1): An Instantiation of the High Level Microprogramming Language Schema S* for the Nanodata QM-1," *Proc. 14th Annual Microprogramming Workshop, MICRO 14*, pp. 124-130, IEEE Comput. Soc. Press (1981).

[Kor75a.] Kornerup P. and Shriver B.D., "An Overview of the MATHILDA System," *SIGMICRO Newletter (ACM)*, Vol. 5(4), pp. 25-53 (1975).

[Kra80a.] Kraley, M. et al, "Design of a User Microprogrammable Building Block," *Proc 13th Annual Workshop on Microprogramming*, pp. 106-114, IEEE Comput. Soc. Press (1980).

[Lan80a.] Landskov D., Davidson S., Shriver B.D., and Mallett P.W., "Local Microcode Compaction Techniques," *ACM Computing Surveys*, Vol. 12(3), pp. 261-294 (1980).

[Lee74a.] Leeman G.B., Carter W.C., and Birman A., "Some Techniques for Microprogram Validation," *Information*

Processing 74 (Proc. IFIP Congress), North-Holland, Amsterdam pp. 76-80 (1974).

[Leh75a.] Lehman M.M., "Microprogramming Trend Considered Dang[illegible]s," *Comm. of the ACM*, Vol. 18(6) (1975).

[Mil71a.] Milner R., "An Algebraic Definition of Simulation between Programs," *Proc. 2nd Int't Joint conf. Artificial Intelligence* (1971).

[MIT79a.] MIT, *Research Directions in Software Technology*, MIT Press, Cambridge, MA. (1979).

[Nan79a.] Nanodata Corporation, *QM-1 Hardware Users Manual: Third Edition, Revision 1*, Nanodata Corporation, Buffalo, N.Y. (1979).

[Oak79a.] Oakley J., "Symbolic Execution of Formal Machine Descriptions," Ph.d Thesis, Dept. of Computer Science, Carnegie-Mellon University (1979).

[Ola82a.] Olafsson M., "The QM-C: A C-oriented Instruction Set Architecture," Technical Report TR81-11, Dept of Computing Science, University of Alberta (1982).

[Owi76a.] Owicki S. and Gries D., "An Axiomatic Proof Technique for Parallel Programs," *Acta Informatica*, Vol. 6, pp. 319-340 (1976).

[Pat76a.] Patterson D.A., *STRUM: Structured Microprogram Development System for Correct Firmware*, IEEE (1976).

[Pat77a.] Patterson D.A., "Verification of Microprograms," Ph.D Thesis, Dept. of Computer Science, UCLA, Los Angeles CA. (1977).

[Rid81a.] Rideout D.J., "Considerations for Local Compaction of Nanocode for the Nanodata QM-1," *IEEE, Proc. 14th Annual Work on Microprogramming MICRO 14*, pp. 205-214, IEEE, Comput. Soc. Press (1981).

[Sto77a.] . tcy J.E., *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory,* MIT Press, Cambridge, MA. (1977).

[Str78a.] Stritter S. and Tredennick N., "Microprogrammed Implementation of a Single Chip Microprocessor," *Proc. 11th Annual Workshop on Microprogramming MICRO 11*, pp. 8-16, ACM/IEEE (1978).

[Tok81a.] Tokoro M., Tamura E., and Takizuka T., "Optimization of Microprograms," *IEEE Trans. Comput.*, Vol. C-30(7), pp. 491-504 (1981).

[Wag83a.] Wagner A. and Dasgupta S., "Formal Semantics of S*(QM-1)," Techical Report (expected), University of Alberta (1983).

Appendix

Declaration Section for S*(QM-1)

```
declaration
    type ls__register = seq [ 17..0 ] bit
    type 18_bits  = seq [ 17..0 ] bit
    type 6_bits     = seq [ 5..0 ] bit
    type 5_bits     = seq [ 4..0 ] bit
    type 3_bits     = seq [ 2..0 ] bit
    type 2_bits     = seq [ 1..0 ] bit
    type bus        = seq [ 17..0 ] bit



    pvar ka, kb, kx, ks, kt : 6_bits
    pvar  cs__addr, ms__addr : bus
    pvar  index_alu_out, alu_out_bus, shft_out_bus, ms_out_bus : bus
    pvar  ksha, kshc : 6_bits
    pvar  alu_op :  tuple
                     kalc : 6_bits
                     cih  : bit
                   endtup
```

```
var  coh : bit

var  sh_end : bit

var  cs_out_bus : bus

var local_store

        : array [ 0..31 ] of 18_bits

        : tuple

              general_purpose   : array [ 0..23 ] of 18_bits

              index             : array [ 0..3 ] of 18_bits

                                     with fmpc : seq [illegible] 0 ] of bit

              general_purpose2  : array [ 0..2 ] of 18_bits

              instruction_reg   :  18_bits

                                :  tuple

                                        c : 6_bits

                                        a : 6_bits

                                        b : 6_bits

                                    endtup

                                :   tuple

                                        opcode      : seq [ 6..0 ] bit

                                        a_parameter : seq [ 4..0 ] bit

                                        b_parameter : seq [ 5..0 ] bit

                                    endtup

          endtup

        : array [ 0..31 ] of 18_bits

             with fcod, faod, fsod, feod : seq [ 4..0 ] bit
```

```
struct    ocal__store__in : selector fcod, faod, fsod, feod : seq [ 4..0] bit
                          union
                             local__store : array [ 0..31 ] of 18_bits
                          endun
     var external_store
          : array [ 0..31 ] of 18_bits
          : array [ 0..31 ] of 18_bits with feia : seq [4..0] bit
          : tuple
               port_register     : array [ 0..7 ] of 18_bits
               operand_source    : array [ 0..7 ] of 18_bits

               base_address      : 18_bits
               field_length      : 18_bits

               interrupt_enable  : array [ 0..1 ] of 18_bits

               alt_base_address  : 18_bits
               alt_field_length  : 18_bits

               interrupt_address : array [ 0..9 ] of 18_bits
            endtup
```

```
struct external_store_output

          : array [ 0..63 ] of 18_bits with feoa

          : selector feoa union

                external_store  : array [ 0..31 ] of 18_bits
                all_zeros       : array [ 0..31 ] of 18_bits

            endun

struct  control_store_data

          : array [ 0..63 ] of 18_bits with fcid

          : selector fcid union

                local_store     : array [ 0..31 ] of 18_bits
                all_ones        : array [ 0..31 ] of 18_bits

            endun

struct  external_store_input

          : array [ 0..63 ] of 18_bits with feid

          : selector feid union

                local_store     : array [ 0..31 ] of 18_bits
                all_ones        : array [ 0..31 ] of 18_bits

            endun
```

```
struct  r_alu

        : array [ 0..63 ] of 18_bits with fair

        : selector fair union

              local_store     : array [ 0..31 ] of 18_bits
              all_ones        : array [ 0..31 ] of 18_bits

          endun

struct  l_alu

        : array [ 0..63 ] of 18_bits with fail

        : selector fail union

              local_store     : array [ 0..31 ] of 18_bits
              all_ones        : array [ 0..31 ] of 18_bits

          endun

struct  shft_in

        : array [ 0..63 ] of 18_bits with fsid

        : selector fsid union

              local_store     : array [ 0..31 ] of 18_bits
              all_ones        : array [ 0..31 ] of 18_bits

          endun
```

```
var f_store                          /* residual control f registers        */
        : array [ 0..31 ] of 6_bits
        : tuple
              fmix  : 6_bits        /* main store input--address/data      */
              fmod  : 6_bits        /* main store output--data             */
              fcia  : 6_bits        /* control store input--address        */
              fail  : 6_bits        /* alu input--left                     */
              fcid  : 6_bits        /* control store input--data           */
              fair  : 6_bits        /* alu input--right                    */
              fcod  : 6_bits        /* control store output--data          */
              faod  : 6_bits        /* alu output--data                    */
              fsid  : 6_bits        /* shifter input--data                 */
              fsod  : 6_bits        /* shifter output--data                */
              feid  : 6_bits        /* external store inpuut--data         */
              feod  : 6_bits        /* external store output--data         */
              feia  : 6_bits        /* external store input--address       */
              feoa  : 6_bits        /* external store output--data         */
              fact  : 6_bits (unused)      /* auxilary action */
              fuser : 6_bits  (unused)     /* user partition  */
              fmpc  : 6_bits              /* mpc pointer     */
              fidx  : 6_bits  (unused)     /* index           */
              fist  : 6_bits        /* global status  */
              fiph  : 6_bits (unused)      /* phantom         */
              g       array [ 0..11 ] of 6_bits
        endtup
```

```
var control_store : array [ 0..40959 ] of 18_bits with cs_addr
  struct cs_addr_source
        tuple
          reg_addr            : array [ 0..63 ] of 18_bits with fcia
                              : selector fcia  union
                                  local_store    : array [ 0..31 ] of 18_bits
                                  all_ones       : array [ 0..31 ] of 18_bits
                                endun
           cs_out_bus          : bus
           index               : array [ 0..3 ] of 18_bits
                                   with fmpc : 2_bits
           index_alu_out       : bus
        endtup
        var main_store : array [ 0..98303 ] of 18_bits

                              with  ms_addr

    struct ms_sink

            : array [ 0..63 ] of 18_bits

               with fmod     /* if fmod > 39 then null operation results */

            : tuple

                  local_store    : array [ 0..31 ] of 18_bits
                  port_register  : array [ 0..7 ] of 18_bits
                  null_operation : array [ 0..23 ] of 18_bits (unused)

              endtup

    struct ms_source

            : array [ 0..63 ] of seq [ 17..0 ] bit with fmix

            : selector fmix union

                  local_store    : array [ 0..31 ] of 18_bits
                  port_register  : array [ 0..7 ] of 18_bits
                  all_ones       : array [ 0..23 ] of 18_bits

              endun
```

```
struct index_alu_l : selector a, b, kx, ka, kb, g_sel : 5_bits
                     union
                       local_store     : array [ 0..31 ] of 18_bits
                     endun
struct index_alu_sink : selector  g_sel : 5_bits
                     union
                       local_store     : array [ 0..31 ] of 18_bits
                     endun

struct index_alu_r : selector a, b, kt, kb, g[8], g[9], g[10], g[11] : 4_bits
                     union
                operand_source   : array [ 0..7 ] of 18_bits
                base_address     : 18_bits
                field_length     : 18_bits
                all_ones         : array [ 0..31 ] of 18_bits
                cntr_  gs        :  18_bits
                ms_out_bus       :  18_bits
                es_out_bus       :  18_bits
              endun
```

```
struct g_sel : selector gspec : 5_bits
                         union
                         g    : array [ 0..11 ] of 6_bits
                         ksha : 6_bits
                         b    : 6_bits
                         ks   : 6_bits
                         kx   : 6_bits
                         endun

struct xalu_op : selector fsel2 : 4_bits
                       union
                         a    : 6_bits
                         b    : 6_bits
                         ka   : 6_bits
                         kb   : 6_bits
                         fmpc : 6_bits
                         null1 : 6_bits (unused)
                         null2 : 6_bits (unused)
                         g    : array [ 0..11 ] of 6_bits
                       endun
```