Technical Report

UCAM-CL-TR-215
ISSN 1476-2986

Number 215

UNIVERSITY OF
CAMBRIDGE
Computer Laboratory

A structured approach to the verification of low level microcode

Paul Curzon

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted May 1990 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Christ's College.

ISSN 1476-2986

Preface

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration. No part of this dissertation has already been, or is concurrently being, submitted for any degree, diploma, or other qualification at any other university.

Acknowledgements

I am grateful to many people who have given their support, advice and time during the period of this research. In particular, I am deeply indebted to my supervisor Martin Richards for his encouragement and stimulating discussion. I am grateful to the members of the Hardware Verification Group in Cambridge, and in particular to Inder Dhingra, Mike Gordon, John Herbert and Jeff Joyce, for their interesting discussion and advice about various things. Many people suggested improvements to this dissertation and gave general advice including Joe Dixon, Andy Gordon, Phil Hunt, Neil Laws, Tim Leonard and Colin Smith. I would also like to thank the members of Room 205, past and present for providing a lively and interesting working environment, and to Martyn Johnson and Graham Titmus for keeping the machines running. My work was supported by the Science and Engineering Research Council and Christ's College, and a High Level Hardware Orion Microcomputer was provided on Alvey grant number ALV/PRJ/SE/034 for which I am grateful. Finally I would like to thank all my friends for maintaining my sanity over the past three years.

Summary

Errors in microprograms are especially serious since all higher level programs on the machine depend on the microcode. Formal verification presents one avenue which may be used to discover such errors. Previous systems which have been used for formally verifying microcode may be categorised by the form in which the microprogram is supplied. Some demand that it be written in a high level microprogramming language. Conventional software verification techniques are then employed. Other methods allow the microcode to be supplied in the form of a memory image. It is treated as data to an interpreter modelling the behaviour of the microarchitecture. The proof is then performed by symbolic execution. A third solution is for the code to be supplied in an assembly language and verified at that level. The assembler instructions are converted to commands in a modelling language. The resulting program is verified using traditional software verification techniques.

In this dissertation, I present a new universal microprogram verification system. It achieves many of the advantages of the other kinds of systems by adopting a hybrid approach. The microcode is supplied as a memory image, but is transformed by the system to a high level program which may then be verified using standard software verification techniques. The structure of the high level program is obtained from user supplied documentation. I show that this allows microcode to be split into small, independently validatable portions even when it was not written in that way. I also demonstrate that the techniques allow the complexity of detail due to the underlying microarchitecture to be controlled at an early stage in the validation process. I suggest that the system described would combine well with other validation tools and provide help throughout the firmware development cycle. Two case studies are given. The first describes the verification of Gordon's computer. This example, being fairly simple, provides a good illustration of the techniques used by the system. The second case study is concerned with the High Level Hardware Orion computer which is a commercially produced machine with a fairly complex microarchitecture. This example shows that the techniques scale well to production microarchitectures.

Contents

Chapter 1

Prologue

Errors in microprograms are especially serious, since all higher level programs on the machine depend on the microcode. When used in life-critical circumstances, an incorrect microprogram can have devastating effects. Furthermore, microprogramming errors discovered late in the design cycle can be very expensive in both time and effort to correct, particularly when the microcode is hardwired within a microprocessor.

Generally, an important design issue when writing microcode is that the program be both fast and compact. The result is that microprograms tend to have far more complicated control structures than normal programs. As suggested in the Orion Microarchitecture Manual [HLH 84]:

> "It is considered acceptable to employ programming 'tricks' at the microcode level since the system performance will be proportional to the number of microinstructions executed."

The algorithms being coded are becoming more complex with little in the way of development tools to aid the microprogrammer. Consequently, errors are easier to make and much harder to detect.

1.1 Discovering Errors in Microprograms

There are several validation techniques which could be employed to discover errors in microcode:

- Testing,
- Program Flow Analysis, and
- Formal Verification.

1.1.1 Testing

Exhaustive testing, though desirable, is unfeasible, even for trivial microprograms, due to the large number of cases which must be tested. Inexhaustive testing is the traditional validation method. Test patterns are used to exercise the program. If it passes all the tests, it is assumed to be correct and released to the public who then proceed to discover further bugs. With inexhaustive testing, confidence that all the errors have been found cannot be attained. Errors can always lie in untested areas. Fixing bugs found after a product has been released can be very expensive and time-consuming. It can also be very annoying for the users of the system. Formal approaches to microprogram testing, which attempt to ensure that the test data selection criteria are sufficient to detect errors, have been suggested to alleviate the problems of inexhaustive testing [Berg 81].

1.1.2 Program Flow Analysis

An alternative way to validate microcode is program flow analysis [Bergeretti 85] [Carré 86] [Foster 86]. This analysis finds certain types of errors, such as two instructions following one another illegally. The basic idea is that various relations are computed for each node in the flow graph representing the code. From these, the relations for the paths through the code can be determined. They can then be used to indicate the presence of different kinds of errors on the paths, depending upon how they are interpreted. The algorithms allow one to be certain that no such errors are present if this is so, though only those of the kind looked for will be detected. Also, they could suggest that correct code contains errors. For example, errors on unexecutable paths might be indicated. The redundant register transfer, where the value transferred is overwritten before being referenced, is one type of error commonly detected by this technique. In microprograms this occurs frequently in non-erroneous situations. For example, a carry flag might be set whenever an addition is performed, even though its value will not always be used.

1.1.3 Formal Verification

Formal verification attempts mathematically to prove the correctness of a program from a formal specification. The proof demonstrates the correctness for all inputs, rather than a chosen few as with inexhaustive testing. The requirement of a formal specification for formal verification also forces the client to resolve ambiguities earlier in the design than would an informal specification.

Some degree of automated help is essential for formal verification. Program verification generally involves the proof of large numbers of simple theorems. This can be slow and tedious, and mistakes are easy to make. The situation is exasperated further when proving microprograms correct, due to the additional complexity of the underlying microarchitectures. Single microinstructions may encode many, often redundant, register transfers. Thus, small microprograms may involve large amounts of detail. With software, it is feasible to verify simple algorithms by hand. This is no longer so with microprograms. Automated systems, on the other hand, are very good at coping with large amounts of detail and doing large numbers of simple tasks. The need for mechanical aid to overcome these problems is illustrated in previous hand proofs. Birman originally carried out a hand proof of the S-Machine

[Birman 74]. However, further errors, missed by the hand proof, were found when the proof was repeated with mechanical aid [Carter 75]. Cohn's machine proof of the first level of the VIPER microprocessor using the HOL system [Cohn 87] found errors in an earlier hand proof. For small examples, hand proofs can be used to find errors in microprograms. With machine assistance, more complex programs can be tackled, and a greater degree of confidence that errors will be discovered is gained. Complete automation is not desirable, however. Much of the advantage of formal verification arises from constructing or guiding the proof, rather than in receiving a yes/no answer as to the correctness of the program. The former forces the verifier to examine the code and the reasons for its correctness. When errors are detected, they will consequently be easier to locate and correct.

In general, program verification systems have been used to verify algorithms rather than production programs. Producing a convincing demonstration of the practicality of microcode verification ought to be more feasible than for software verification because of the nature of microprograms. Indeed, verification systems have been used in the verification of production microprograms for commercial computers [Carter 78a] [Patterson 78] [Damm 85b] [May 87] [Crocker 88]. Microprogram verification is more feasible for several reasons.

- Microprograms deal with simple datatypes (bitstrings).
- The microcoded algorithms tend to be numeric.
- The code is short compared to software.
- The microarchitecture is well-defined and fairly simple, leaving less doubt as to the semantics of microinstructions.
- The specification of the code is unlikely to be frequently changed. Firmware does not evolve in the manner of software.
- Microcode does not normally use recursive procedures.

A formally verified program cannot be assumed to be correct. It may still contain errors due to mistakes in the specification, proof, etc. Consequently, formal verification should be used alongside testing and program flow analysis as a tool for finding and correcting errors.

1.2 The Problem

In the previous section, I described the techniques available for detecting errors in microprograms. It is with formal verification that this dissertation is primarily concerned. I have suggested that formal verification of production microprograms for commercial microarchitectures is feasible. A number of problems remain.

1.2.1 Complex Microarchitectures

If a microprogram verification system is to be a useful tool for commercial architectures, the complexity arising from large amounts of detail due to the microarchitecture must be dealt with. Machine assistance helps greatly in this way. However, if the verifier is to guide the proof, the techniques used must deal with this complexity at an early stage and in such a way that the output is understandable. If this is not done, the verification task will become intellectually unmanageable.

1.2.2 Control Structures

Complexity may also arise due to the volume of code being verified. Microprograms implementing commercial microarchitectures will inevitably be large. They will also tend to have complex and tangled control structures. The use of goto statements in high level languages has long been considered bad programming practice. It generally produces unreadable code which is consequently liable to contain errors and is difficult to verify. However, jumps proliferate in microprograms due to the absence of high level control structures and the need for the microcode to be fast and compact. Ideally, for validation purposes, code should be structured and modular, allowing small portions of code to be independently validated. For microprograms this is unlikely to be the case.

1.2.3 Universality

An important feature of any firmware engineering tool is that it be universal in the sense that it may be used with microprograms for any microarchitecture, without the system needing to be rewritten. To achieve this, the system must use a formal description of the microarchitecture semantics. Obviously, it is a great advantage if one set of tools can be used for any microarchitecture. The advantage is greater than it might first appear, since a particular microarchitecture may only have a single microprogram written for it. For this reason, it may not be worthwhile developing a new set of tools. Furthermore, microarchitectures often tend to evolve with the microprogram during the firmware development cycle. If tools are to be useful during this cycle, they must also be capable of adapting. If parts of the tools need to be rewritten whenever such a change is made, either the tools will always be behind, and of little use, or the development cycle will be greatly slowed to allow the tools to keep up to date. This could be disastrous for a product. With technology rapidly advancing, delays in the development cycle could render a new architecture out of date before its release. An additional advantage of using universal tools is that it forces the designers to write a formal specification of the microarchitecture.

1.2.4 Self Modifying Code

Self modification, in which the code ceases to be a static object, is a problem unique to the verification of low level code. In the past this has not been a problem since microcode has tended to reside in read-only control stores. However, with the advent of cheaper memory, writable control stores are commonly used allowing data and program regions to be intermixed. Although the intentional use of self modifying code may be rare, the microcode in a writable control store could inadvertently overwrite itself. A verification system must be capable of coping with self modifying code. At the very least, it should be possible to verify that the code does not modify itself.

1.3 The Contribution of this Dissertation

In this dissertation I present a new microprogram verification system which overcomes the problems described in the previous section.

- I describe ways of reducing the complexity arising from commercial architectures at an early stage in the verification process which are amenable to automation.
- I show how the microprogram can be split into intellectually manageable portions that can be verified independently without the microcode being written that way, overcoming the problems of verifying large bodies of code.
- I show how extra formal documentation can overcome the problems of complex control structures and also give additional help in the firmware development cycle.
- I show that the advantages of the various previous kinds of systems can be achieved by adopting a hybrid method.
- I describe universal verification tools which both ease the problems of *post facto* verification and can also be used as an aid in the development process.
- I show that microprograms and microarchitectures do not necessarily have to be designed with verification in mind to make it feasible.
- I suggest that the techniques can cope with microcode which could, intentionally or otherwise, modify itself.

1.4 Terminology

There is some confusion in the literature over the terms "host" and "target". Some authors use the term "target" in the sense of retargetable tools, i.e., tools that can be used for different microarchitectures. Thus, "target" is used to refer to microarchitectures. The term "host" is then used to refer to the macroarchitecture. Other authors take the target to be that which is aimed for, i.e., that which is to be implemented—the macroarchitecture. The host is then the microarchitecture since it plays host to the microprogram. Throughout this dissertation, I use the latter meanings, since they are predominant in the firmware verification literature, and seem more natural.

- **Target** ≡ Macroarchitecture
- **Host** ≡ Microarchitecture

Chapter 2

Microprogram Verification Systems

In this chapter I describe microprogram verification systems from the literature and compare the various approaches adopted. I then outline SPOOK: the new system which is the subject of this dissertation, and suggest the advantages that it has over previous systems. I also outline the remainder of the thesis.

2.1 Previous Systems

A number of different formal microprogram verification systems have been described in the literature. These have fallen roughly into three types, according to the form the microprogram takes:

- High Level Microprogram Verification Systems,
- Memory Image Verification Systems,
- Assembler Level Verification Systems.

The advantages and disadvantages of each type of system are discussed below. The individual systems which fall into these categories are also described. Descriptions of some machine code verification systems are given since many of the techniques and problems are related to those of firmware verification. Dasgupta [Dasgupta 88] gives a more detailed overview of many of the systems described.

2.1.1 High Level Microprogram Verification

High level microcode verification requires that the code be written in a high level microprogramming language. Standard software verification techniques are then used to verify this high level code, by, for example, a Hoare proof based on an axiomatic semantics. This approach has several advantages.

- Writing high level code is much easier than writing low level code, so fewer errors occur in the first place.
- The code can be produced hand in hand with a proof of its correctness. This is generally accepted to be simpler than *post facto* verification.
- The programs are easier to document.
- Structured programming techniques can be enforced, simplifying the verification task.
- Axiomatic proof techniques require that the code is formally documented with annotations.

However, there are disadvantages.

- By their very nature microprograms are machine specific. Any particular high level language is unlikely to be suitable for the wide variety of host architectures which exist. Moreover, the cost of producing software engineering tools for a machine dependent language is liable to be prohibitive if they are only to be used to implement a single macroarchitecture, as is commonly so. This would especially be a problem when the microcode was designed in conjunction with the microarchitecture itself. The tools would then have to be constantly rewritten to keep up with the design changes.
- The efficiency of microprograms is crucial. Compilers are unlikely to produce code which is as efficient as hand optimised code. This is especially true when compiling to microcode, owing to the additional problems of packing microinstructions. There are always likely to be time critical tasks for which hand produced code is required.
- The compiler itself may introduce bugs into the code. Some work on compiler correctness has been performed (see for example, [Moore 88] [Joyce 89c]), though it is still a research problem. If an optimising compiler is used, the compiler will be more complex and more likely to introduce errors.
- If changes are made to the resulting code output by the compiler, for example to optimise it, the proof of correctness will be invalidated. There is often a great temptation to hand optimise code in this way.

STRUM

Patterson's STRUM system [Patterson 79] [Patterson 81] is one such high level microprogramming verification system and is based on the London Software verification system [Igarashi 75]. Microprograms are written in a Pascal like language which is defined by an axiomatic semantics allowing Hoare proofs to be conducted. The system has been used to verify an emulation of the HP2115 on the Burrough's D-Machine. This consists of a 1700 line source program which compiles into a 940 microinstruction microprogram after optimisation [Patterson 78] [Patterson 81]. The problems of producing efficient code were avoided because the system and language were designed around one particular host architecture. In fact the optimised STRUM emulation was faster than an emulation programmed independently in assembly language. To use the system for a different host architecture would require the rewriting of the declaration and code generation parts of the compiler. Since the language was designed specifically for the D-Machine, it is not clear that the results would be so impressive for different host architectures.

$S^\star$

The machine independent language schema $S^\star$ [Dasgupta 80] was developed to overcome the problems of using machine independent high level microprogramming languages. $S^\star$ forms a family of languages for machine design which also includes $S^\star_A$—an architecture description language [Dasgupta 81].

The schema is instantiated for a particular host machine to give a machine dependent language. This allows a well-structured microprogram to be produced, as with other high level languages. Additionally, it allows the microprogrammer to program using the particular microoperations available and indicate which should be packed into single microinstructions. Alternatively, code optimisation and compaction may be left to the compiler. Consequently, the efficiency of the implementation is not compromised.

Dasgupta and Wagner provided an axiomatisation for the machine dependent instantiation $S^\star$(QM-1), for the QM-1 architecture [Wagner 83] [Dasgupta 84]. This has a two level control store where by the microinstructions are interpreted by an even lower level program. It also makes use of residual control in which part of the control word is stored in registers. A relatively small and uniform set of proof rules were discovered for $S^\star$(QM-1). However, as they state, "the proof rules are far too cumbersome to be used in practical firmware design" [Dasgupta 84]. This is largely due to the many side effects which arise from attempting to divide microinstructions into distinct microoperations. The solution suggested was to use less complex architectures in cases where verification was desirable. A hand proof of the QM-C multiplication routine was described.

Recently Dasgupta has been involved in producing an axiomatic description language for describing computer architectures, $S^\star$M [Dasgupta 86], based on $S^\star$. The language allows separate units within the architecture to be described in a modular, structured fashion in the form of preconditions and postconditions, so amenable to this form of verification. The timing characteristics of each module may be specified in the form of guards. These include conditions which, if true, cause the activation of the module at the start of the indicated clock phase. Modules are thus activated non-procedurally.

AADL/S*

The work of Dasgupta and Wagner has been furthered by Damm [Damm 84a] [Damm 85a] [Damm 86]. The AADL/S* system combines the architecture independent language schema S* with the architecture description language AADL. AADL is an axiomatic description language similar to S*M, though is procedurally based. Like S*M it permits the specification of timing behaviour.

The operations of an architecture, A, are described using AADL. This formally defines the syntax and semantics of a machine dependent language S*(A), which may then be used to program on the architecture. The AADL specifications for operations are converted into substitutions which are used to form proof rules for that operation. Microprograms are thus written in S*(A) for a given microarchitecture A, described in AADL. The facilities of S* again allow the programmer to indicate which microoperations are to be packed into single instructions. Damm has extended the proof rules to allow proofs that microoperations are dynamically disjoint. This means that they are prevented from being concurrent only if they will use the same resources. This contrasts with static disjointness, in which microoperations are considered to interfere if there is any possibility of a resource usage clash, even if run time conditions would prevent it [Damm 84b] [Damm 88]. Unfortunately the proof rules are very complex.

AADL is also used to specify the higher level architecture being implemented. Preconditions and postconditions are derived for each module's specification and so can be used in proofs. This allows the system to be split into a number of hierarchical layers which may be proved independently.

The methodology was applied to the design of an emulation of the NOVA–1200 computer on the MICRODATA 1600; a commercially available architecture which allows limited microcode self modification and uses residual control [Damm 85b]. In the course of this exercise, an I/O handling error was detected in the original code upon which the emulation was based.

INMOS

The system used by INMOS to formally derive the microcode for the IMS T800 floating point unit is based around the Occam language [May 87] [Shepherd 88]. This has a rich formal semantics in the form of transformational laws [Roscoe 86]. Using these laws, Occam programs may be transformed into other equivalent ones. This notion is the basis of the system. Rather than compiling a high level program into a low level one, the microprogrammer uses the laws to transform it into one. In this way, the problems of compiler correctness and producing efficient code are avoided, though the microprogrammer does much more work.

The specification is first refined into a high level Occam implementation using the methods suggested by Gries [Gries 81]. This high level Occam program is then transformed into a low level one, close to the form of the assembly language. This involves converting test conditions, arithmetic and logical unit operations and control sequencing to forms available on the microarchitecture. The transformation process is not fully automated; instead the system applies transformations as directed by the user. A pattern matcher then converts the low level Occam into assembler source, which may be assembled into the bit pattern code.

The system was still under development during the coding of the IMS T800 floating point unit. Therefore, the above process was not strictly adhered to. Instead, for most instructions two full versions of the code were written—one high and one low level. The low level code was transformed into the form of the formally verified high level program, ensuring its correctness. A second design was produced in parallel using conventional design techniques. The transformational design was ultimately used, however, as it overtook the conventional design, illustrating the success of the system.

2.1.2 Memory Image Microprogram Verification

Other methods of microcode verification take the microprogram in its bitstream form. It is treated as input data to an operational description of the host machine. The verification involves symbolically executing the host machine description, stepping it forwards through some number of microcycles and comparing the resulting symbolic state with that expected for the target machine.

These methods avoid the problems of high level coding.

- Any microcode can be verified, no matter which method was used to produce it.
- Programming methods which compromise the efficiency of the code do not have to be used.
- The actual memory image is verified, so there is no problem of compiler correctness.
- If the description of the host program is given in an executable language, it can be used to perform simulations. This is a major advantage if the microprogram is being developed in tandem with the architecture.
  - It allows normal testing to be conducted without needing the underlying hardware to be available.
  - It means that a special simulator need not be written for each new architecture.
  - It means that changes to the architecture will not require changes to the simulator, only to the architecture's formal specification.

However, there are still disadvantages.

- The verification is performed *post facto*. No help in the development process is provided.
- The microcode is treated as data. Consequently, structuring the proof is less natural, since the higher level concept of the code being a program is lost.

- Since the code can be produced by any means, it is liable to be completely unstructured, involving lots of jumps and possibly self modifying code.

- In general, these systems do not require any documentation of how the code implements the functional specification. The functional specification itself is all that is used.

- Code that is relocatable cannot be verified since a memory image is used as the microprogram source.

MCS

MCS [Carter 77] [Carter 78a] originated from the work of Birman [Birman 74]. It is an automated system which was very successfully applied symbolic simulation techniques to production microcode. Macro and micro level specifications are written in LSS—a Language for Symbolic Simulation [Joyner 78] [Carter 79]. A simulation relation is also given. This describes relationships which should hold between the resources of each level machine at particular corresponding control points. Each description is then symbolically executed between these control points, and it is proved that the resulting states correspond, according to the simulation relation. Loops in either host or target code are broken by inductive assertions [Leeman 74]. The automated system provides a problem reduction framework [Birman 76] [Joyner 76] based on the ideas behind the LCF [Gordon 79] theorem prover. The problem of proving simulation is initially set as a goal. This is then refined into simpler and simpler subgoals until it is proved.

The technique was first used to verify code for the S-Machine architecture—a simple hypothetical machine. The microarchitecture was vertically microprogrammed and the proof was performed by hand. Three errors were found in the microcode [Birman 74]. The proof was later repeated using MCS and an additional error was found [Carter 75]. For this automated proof, additional control points were added. However, this involved rewriting the target specification to mirror the microcode control structure so that points in the specification corresponded to control points in the microprogram. Further hand proofs were performed for two implementations of the S-Machine on a horizontal microarchitecture. The first was a naive implementation whereas the other overlapped instruction fetching and execution [Carter 77]. A horizontally microprogrammed extended version of the S-Machine was also proved [Leeman 75]. This implemented a division instruction and allowed up to three levels of indirection. MCS was also used to verify production microcode—that implementing a version of the NASA Standard Spaceborne Computer-2 which supports the 86 standard System/360 instructions and includes a single I/O channel. A number of errors in the code were discovered during the verification attempt [Carter 78a] [Carter 78b]. In connection with this proof, some research was performed using MCS to verify communication protocols [Brand 78]. In particular a simplified version of the HDLC protocol was verified [Brand 82].

SDVS

The SDVS approach [Crocker 80] [Marcus 84] is similar to that of MCS. In both systems the host, in conjunction with a bit representation of the microcode, and the target specifications are symbolically executed between user supplied stopping points. In SDVS the host and target specifications are given in the programming language ISPS. The system then converts these to a set of temporal logic formulae called "state deltas". These describe state changes in the machine over a time interval. Conceptually, one state delta is created for each ISPS statement. Points in the target program which correspond to points in the execution of the microprogram may be identified. Symbolically executing the sets of state deltas between these points give new state deltas which describe the state changes between the points. Case analysis is performed at branch points, giving separate state deltas incorporating appropriate path conditions. Typically, the stopping points will correspond to the boundaries of a macrocycle and the state deltas will represent the state changes caused by the target level instructions. As with MCS, the proof cannot be subdivided according to the structure of the microprogram unless such subdivisions correspond to points in the target program. This will generally not be so unless the target description is written in a way that mirrors the microcode, or vice versa. The correctness proof involves showing that the target level state deltas are implied by the host state deltas together with the microcode and a map relating target level resources to those of the host.

Initial experiments with the system involved work on a simple GCD Machine [Levy 84], the H-Machine—a simplified version of the AM2901 [Marcus 84], and the Fault Tolerant Spaceborne Computer—a 32 bit machine intended for maintenance free survival in space [van-Mierop 78]. Recent work has included the verification of the BBN C/30, which is used in the packet switched nodes of the Defense Data Network and contains almost 5000 words of microcode [Crocker 88]. Also, a reverification of the FM8501 microcode, originally verified using the Boyer–Moore system described later, has been performed [Crocker 88].

MIDDLE

The verification method developed by Budkowski et al. [Budkowski 78] also takes a bitstream version of the microprogram. Unlike the other approaches which start with low level code, the microprogram is not just treated as data to the host machine. Instead, it is combined with a description of the hardware to give a microprogram written in the specially designed language MIDDLE [Dembiński 78a] [Dembiński 78b]. This may be transformed to a register transfer level program using syntactic transformations and simplification. The resulting program may then be verified using any conventional software verification technique, not necessarily symbolic simulation.

The MIDDLE program has the same structure as the original microcode. Unless the microcode was written with verification in mind, it is likely to contain many jumps and have a fairly scrambled control structure. Such a control structure greatly complicates the task of verification and does not help make the program intelligible.

Transformations are suggested which could be applied to convert the program to a more suitable form for verification, though the complex control structure will remain [Dembiński 83]. Also, as with the INMOS method these transformations are chosen by hand. The situation is exasperated further, due to the simplistic method used to convert the bitstream version to MIDDLE code. The calculation of possible next addresses is performed by inspection of only the current microinstruction. No account is made of the effect of instructions executed at earlier points on paths through the instruction. This means that many unexecutable jumps may be included in the resulting program, rendering it unnecessarily complicated. This could be a serious hindrance if used on complex architectures.

The method was used to prove several arithmetic microprograms for a floating point arithmetic and logical unit designed at the Warsaw Technical University. These proofs were performed by hand, as the system was not mechanised. The software verification technique used to verify these programs was an algebraic method [Blikle 76]. The semantics of individual MIDDLE commands was described by input-output relations, which were composed to give a relation for the whole program. This was then manipulated algebraically by applying a set of transformations to it until it had the form of the target description.

HOL

HOL [Gordon 87] [Camilleri 86] and its forerunner LCF-LSM [Gordon 81a] are LCF [Gordon 79] style theorem proving systems used for hardware verification. The underlying logic of the HOL system is higher order logic which has also been used for hardware verification and specification by Hanna and Daeche [Hanna 86]. In these systems, a structural specification of the implementation of a device and its functional specification are given in the logic. A theorem giving a correctness statement relating these descriptions, with respect to suitable abstractions, is then proved using the theorem prover.

In addition to the proofs of other hardware devices, the HOL system has been used to verify a microcoded machine—Gordon's computer [Joyce 86] [Joyce 88b] [Joyce 89a]. This is a simple toy machine, designed as an example of hardware verification. The proof was originally performed using the LCF-LSM system [Gordon 81b]. The Tamarack microprocessor, which is a version of Gordon's computer, has since been fabricated as a CMOS chip [Joyce 88a] [Joyce 88b]. Recently, a verification of microcode for an extended version of the Tamarack microprocessor, Tamarack-3, has been performed [Joyce 89b]. This computer performs asynchronous interactions with the external environment using a handshaking protocol.

The HOL system has also been used in the verification work performed on the VIPER microprocessor. This is a commercially produced microprocessor developed using formal methods [Cullyer 88]. It is specified in a sequence of decreasingly abstract levels. A HOL proof has shown the equivalence of the top two of these levels [Cohn 87]. Some work has also been performed on a proof of the next level [Cohn 88]. The former proof was originally performed by hand. The HOL proof highlighted discrepancies in this proof, though showed it to be basically sound. The top level specification (the target specification) is a functional specification giving transformations between high level machine states. This was shown equivalent to an implementation on the next level—the major state machine. This implementation executes a series of events which change the machine state. Thus, the major state machine is similar in nature to a microcoded implementation of the functional specification with respect to the proof. The sequences of events define a state transition graph, which takes the place of a program. Execution of events is similar to executing microinstructions. The proofs of Gordon's computer and the VIPER major state machine follow the same basic form. The implementation is symbolically simulated to produce theorems giving the effects of each path through the microcode. This is done by repeatedly expanding the definition of the host under suitable path conditions. The correctness statement is then proved by case analysis of the possible path conditions.

A disadvantage of using HOL is that the logic is not directly executable, preventing simulations of HOL specifications from being performed. This is useful for increasing confidence in the correctness of the specifications. Methods for converting a subset of the logic into an executable form have been suggested, however [Camilleri 88]. A further disadvantage is that HOL requires significant mathematical ability to drive. It is more suited to use by a team of verification specialists rather than by actual programmers.

VERIFY

The VERIFY system [Barrow 84b] [Barrow 84a] also evolved from the LCF–LSM approach for verifying digital hardware designs. As with the LCF–LSM and HOL work, the approach is geared to verifying hardware designs in general, rather than microprograms. The emphasis is placed on modularising the proof with respect to the hardware description, rather than the microcode. The user supplies a structural and behavioural specification of the hardware. The structural description is modular, with equations describing the inter-connections between modules for each state variable and output. The structural specification is converted into a behavioural one, and equivalence with the supplied behavioural specification is shown. For a microprogrammed implementation, this involves showing that a behavioural specification of the macroarchitecture is equivalent to an unfolded version of the microarchitecture specification. This is done by performing case analysis to produce all paths through the code, and then producing equations giving the effects of each path on each state variable and output. These are then shown to be the same as for the macroarchitecture. The VERIFY system is largely automatic and is implemented as a PROLOG program. It has been used to redo the proof of Gordon's computer, together with proofs of other hardware examples.

The Boyer–Moore Theorem Prover

The Boyer–Moore Theorem Prover [Boyer 79] has also been used as a hardware verification system in a similar way to HOL [Hunt 85]. Structural and behavioural specifications are written in the Boyer–Moore logic together with a correctness statement. This statement is then proved by unrolling the definitions in the specifications and performing case analysis over the paths through the microcode. Unlike the HOL system the Boyer–Moore Theorem prover is intended to be an automatic theorem prover. However, in practice the verifier must guide the system by first proving intermediary lemmas, so knowledge of the heuristics used by the prover is required. A further difference is that the Boyer–Moore logic is executable, allowing specifications to be tested.

The system has been used to prove the correctness of the FM8501 processor. This is a 16 bit microprocessor, similar in size and complexity to a PDP-11 which was specifically designed as a verification case study. The microcode in the FM8501 was very simple, however, consisting of only 16 microwords. A similar though extended machine, FM8502, was also verified.

2.1.3 Assembly Language Verification Systems

Some systems verify assembly language code. Typically, the semantics of the assembler instructions are specified by giving a translation of each into some modelling language. This contrasts with the bitstream methods in which a description of a particular host architecture is given in a hardware description language of known semantics. The translation is used to convert the assembly code into a program in the modelling language. This can then be verified using standard program verification techniques.

Practical systems using this approach have been mainly for machine code proof rather than for microcode proof. This is because microinstructions are often horizontal and consequently the translation process is more complex. This problem is partly overcome by splitting the microinstructions into microoperations. Translations are then given for these individually, though as noted from the QM-1 work, splitting microinstructions into individual microoperations is not always straightforward.

Verifying the code in this way gains some of the advantages of each of the previous methods.

- The code is still written at a low level, so the efficiency of the microcode is not necessarily compromised.

- The problems of the compiler introducing errors are largely avoided. Even though an assembler may introduce errors in the same way as a compiler, this is less likely to be a problem. Assemblers are far simpler than compilers, so are less likely to contain errors. Verification of an assembler would be more practical than for a compiler.

- The code is converted to a precise human readable form. This can be checked, allowing bugs to be spotted at an early stage.

- The program is not just treated as data. Consequently, its structure is not lost to the proof effort.

Disadvantages remain.

- The verification is still performed *post facto.*
- Assembly languages tend to be unstructured, complicating the verification task.
- New problems due to the assembly language, such as aliasing of names, may arise.
- The translation to the modelling language is performed either by hand or by programs specific to individual assembly languages. Errors can be introduced at this point. This would only be a problem if errors in the assembly code were nullified by the translation process. Other errors would just cause the proof attempt to fail so this problem is less critical than for compiler correctness.
- The division of microinstructions into microoperations introduces new problems such as side effects.

Ramamoorthy and Shankar

Early work of this nature with respect to microcode was conducted by Ramamoorthy and Shankar [Ramamoorthy 74]. They were concerned with showing the equivalence of microprograms. This is of use, for example, to determine whether two emulations are equivalent. They only considered simply structured code with no loops and at most a two-way branch in any microinstruction. The method involves first converting the microoperation assembler version of the microprogram into atomic microoperations from a standard set. The two microprograms are then split into separate paths, the path conditions for which are determined. Corresponding paths from the two microprograms are paired, equivalent variables from the machines having been identified by the user. Separate sequences of microoperations are extracted from these pairs, for each pair of corresponding outputs of interest. Symbolic manipulation techniques are then used to show that these outputs are given the same values. The microprograms may also be shown to be correct by comparing the symbolic value given to each output with the specification.

Brioschi and Scaini

Brioschi and Scaini also proposed a method for proving microcode correctness which takes assembler input [Brioschi 76]. Rather than performing a translation from the assembler code, however, it is symbolically interpreted directly. A path analysis is first conducted and the symbolic interpretation performed on each path separately. The resulting states are then shown to be identical to that required by the target specification. The method was not mechanised, and was only used for small examples on a specially designed simple machine.

Maurer

The assembler approach is basically that adopted by Maurer to prove the machine coded flight director program for the Litton C4000 airborne computer [Maurer 76]. This is a simple program with no loops or subscripted variables and only a single subroutine. The proof does deal with overflow conditions, the way the code performs approximate calculations, and scaling problems with fixed point values. Maurer also addresses the problems of potentially self modifying code [Maurer 74]. An IBM 370 assembly language verifier has been produced, and used to verify programs implementing simple algorithms such as Euclid's GCD Algorithm [Maurer 77]. Simple algebraic equivalents were given for each machine code instruction. A variation of Floyd's Inductive Assertions method was then used to prove the code. This proof was computer aided.

SPADE

Clutterbuck and Carré also use this basic approach for verifying macro level assembly language programs [Clutterbuck 86] [Clutterbuck 88]. In their SPADE system, the algebraic modelling language FDL is used. The transformation into FDL is performed by a translator for each assembly language considered. Once the assembly program has been converted into FDL, the SPADE program analysis tools may be used. These include program flow analysis tools (such as a data flow analyser) as well as tools for formally proving program correctness. In the translation to FDL any structure that the code may have had is lost. The flow analysis tools may be used to rediscover it. The verification method is basically Floyd's [Floyd 67]. The specification is given in terms of preconditions and postconditions. Verification conditions are then generated for each path through the code. In addition to a full program proof, a "shallow" proof that certain run time errors such as overflow do not occur can be separately performed.

Source level restrictions are applied to the code to make the verification task more tractable. This means that a suitable subset of the source language must be identified, and programming is done very much with verification in mind. Using the disallowed techniques is deemed to be bad programming practice and incompatible with safety-critical applications.

The SPADE system is being used commercially to validate both low and high level software. It has been used to verify specially written programs for the Intel-8080. The code was written in SPADE-8080 - the "safe" subset of the Intel-8080 language. A more impressive demonstration of the system is the proof of the LUCOL Z8002 assembly code modules for the fuel control unit of the Rolls-Royce RB211-524G jet engine [O'Neill 88]. As a consequence of the formal verification process, the code of one module was altered to cope with extreme inputs and the informal documentation of 10% of the modules was revised.

2.2 The SPOOK System

In this dissertation I introduce a new microprogram verification system—SPOOK, an overview of which is shown in Figure 2.1. The SPOOK system is a hybrid of the previous systems, and so it gains many of the advantages of each. The microprogram is initially taken in the form of a memory image. This is transformed into a high level program in a modelling language as with the MIDDLE and assembler systems. Unlike these systems, however, the resulting high level program is structured and modular, gaining the advantages of software verification techniques. This is achieved by making the user supply extra documentation of the code in the form of a structured and modular state transition graph for the microprogram.

The user of the system supplies four things:

- **The Host Program**—The Microarchitecture,
- **The Skeleton**—A State Transition Graph of the Microprogram,
- **The Target Specification**—The Macroarchitecture, and
- **The Microprogram**—A Memory Image.

The host program, target specification and microprogram are provided in some form to all memory image systems. The need for the skeleton is unique to the SPOOK system. The SPOOK language, which is a simple imperative language derived from BSPL [Richards 86], is used to describe each of the above. In addition it is also used as the modelling language for the high level version of the microprogram that the system produces. Chapter 3 deals with the syntax and semantics of the SPOOK language. Whilst a familiarity with the language is a prerequisite for the remainder of the dissertation, a detailed understanding is not necessary on a first reading.

2.2.1 The Host Program

The host program provides a formal description of the microarchitecture as a clocked finite state machine with a single universal clock. It describes the effects of a single clock cycle on the state and outputs of the machine given the values on the inputs for that cycle. This model is obtained by enforcing a strict structure on the program. It is the use of a host program that allows the system to be universal with respect to the microarchitecture. Only a new host program need be provided to use the system with a different microarchitecture. The use of the host program is described in detail in Chapter 4.

Collapsed Host Programs

Information which is known about the state at a particular instant of time can be used to produce a collapsed host program. These are specialised versions of the host program, which describe the effect of running the machine for one clock cycle from a state satisfying the given condition. They are equivalent to a host program for a

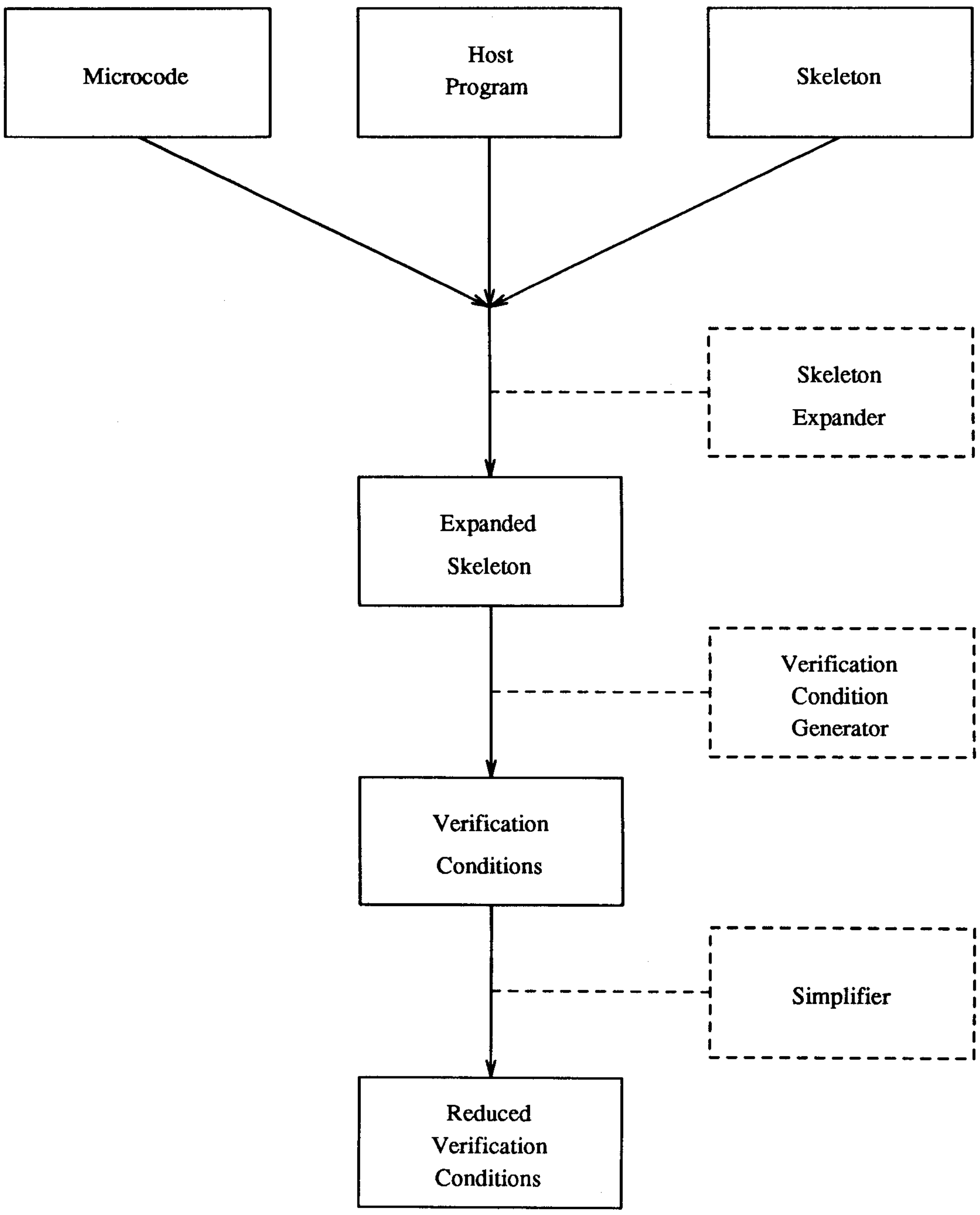

Figure 2.1: An overview of the SPOOK system

machine identical to the original, except with the known values hardwired in. A host program is collapsed by substituting in known values for state variables and then performing constant folding and rewriting. This collapsing mechanism is used in the verification process, reducing the complexity at an early stage. It also provides a useful tool to aid the understanding of microinstructions, giving a concise and formal description of their action. It is one of the major features of the system, and is also described in Chapter 4.

2.2.2 The Skeleton

The skeleton is formal documentation, provided by the user, giving the state transition graph of the microprogram. Such information is often given informally in the form of a diagram. By making the user provide it in the form of formal documentation, the information can be used as part of the verification process. Also, this documentation can itself be verified, ensuring that it is consistent with the microcode. In the SPOOK system, high level language constructs are provided to allow the skeleton to be structured and modular. This is a suitable form from which to produce the high level microprogram. It should be noted that the original microcode does not need to be structured to allow the skeleton to be. The skeleton can be a cleaned up version of the original code.

Expanding The Skeleton

The conditions found in the skeleton, which characterise the nodes in the state transition graph, are a possible source of information to seed the production of collapsed host programs. A collapsed host program can be produced for each node in the state transition graph given by the skeleton. If the nodes in the skeleton are replaced by their corresponding collapsed host program a high level version of the microprogram is obtained. State changes are made according to the collapsed host program and, thus, according to the semantics of the microarchitecture. The program has the modular and hierarchical structure of the skeleton however. Extra legality assertions are also incorporated into the expanded skeleton to allow the validation process to ensure that it is equivalent to the original microprogram. In addition to allowing software verification techniques to be used to verify the microprogram, the expanded skeleton is useful in other ways. It provides documentation of the microprogram, giving a concise, formal and more understandable version of it. Furthermore, simulations could in principle be performed on it using a SPOOK interpreter.

The provision of the skeleton is the most significant difference of the SPOOK system from previous microprogram verification systems. Its use is described in detail in Chapter 5.

2.2.3 The Target Specification

The target architecture is described in the form of a partial correctness specification. A precondition indicates conditions which should hold before the execution of the microprogram. A postcondition then gives the final state. The partial correctness specification states that if the precondition is true before the microprogram is executed, then provided the program terminates, the postcondition will be true afterwards. Proofs of termination are beyond the scope of this dissertation.

The target specification is given as part of the skeleton. Annotations are also placed throughout the skeleton to aid the verification and provide formal documentation of the code. The verification process ensures that the annotations are consistent with the code. The skeleton is modular and structured. Each module can be verified independently and is given a separate partial correctness specification. The target specification is described with the skeleton in Chapter 5. Since this kind of specification is normally used in software verification, only a brief discussion is given.

2.2.4 The Microprogram

The microprogram is input to the system as a memory image so the actual code that runs is verified. It need not be written with verification in mind and does not need to be structured or modular. If the code is held in read only memory, then the microprogram may be given as part of the host program. This occurs, for example, with Gordon's computer. Alternatively, if it is held in random access memory, as in the High Level Hardware Orion computer, it is just part of the initial state and so is given as a precondition in the target specification.

2.2.5 Software Verification Techniques

The expanded skeleton could be verified using any software verification technique, since it is just a high level program. The SPOOK system is based on Hoare logic [Hoare 69]. The SPOOK language, in which the expanded skeleton is given, is defined by an axiomatic semantics. The target specification is given in the form of a partial correctness specification, consisting of a precondition and postcondition. The proof is performed by logically deducing the partial correctness specification from the axioms and rules of inference given in the axiomatic semantics, together with those for the predicate calculus. This process is mechanised in the form of a verification condition generator based on the semantics. This produces a set of predicate calculus statements, the truth of which implies the correctness of the original microprogram. A simple, goal based, semi-automatic theorem prover is then used to prove the majority of the verification conditions. The onus is left with the user to prove any that cannot be proved automatically by the system. These techniques are based on those from the software verification world. The precise way they are used within the SPOOK system are described in Chapter 6.

The theorem proving tools, and also those used when manipulating the host program and skeleton, were not rigorously derived in any formal theory and so it is possible that mistakes could have occurred. The system is just a prototype, designed to illustrate the techniques described. As such, this lack of security is not a major concern. If greater security is required suitable foundations could in principle be developed. Since we are concerned with discovering errors in microprograms in this dissertation, the worst effect of such a mistake in the tools would be that microprogramming errors go undetected. This would be unfortunate, though would not render the tools useless, especially if other errors were detected. Formal verification should be used in conjunction with other validation techniques to increase the likelihood that all errors are found.

2.2.6 Case Studies

Two case studies have been performed to illustrate the use of the system. The first involves the verification of the microcode for Gordon's computer [Gordon 81b], an example which has been used by other verification teams as discussed earlier. It is a simple toy computer designed as a verification test case. Due to its simplicity it is useful for illustrating the techniques used in detail. In particular the verification shows how an unstructured microprogram can be converted into a series of structured and independently verifiable modules. This case study is described in Chapter 7 and provides a good illustration of the way the system is used.

The second case study, described in Chapter 8, concerns the HLH Orion Computer [HLH 84]. This has a fairly complex microarchitecture and uses commercially produced microcode, neither of which were designed with verification in mind. Only a small fragment of the the microcode was verified—that for a single AND macroinstruction. The example does show that the techniques scale up well to a commercial architecture. Also, the advantages of a skeleton are highlighted by the complex pipelined microinstruction sequencing used by the Orion microarchitecture.

2.2.7 Advantages of the SPOOK System

By combining the techniques used by the other types of systems, SPOOK gains many of the best features of each. The system has many of the advantages of the memory image verification systems.

- The code may be hand produced or hand optimised, so efficiency is not forfeited for the sake of correctness.
- There is no problem of compiler correctness.
- The semantics of the microarchitecture is given in the form of a formal description. This means
  - the system is universal with respect to the microarchitecture, in that no part of the system need be rewritten to change hosts, and
  - a universal simulator could be written allowing the code to be easily tested.

Unlike the other memory image systems, the microprogrammer's knowledge of the intended control structure of the program is not lost to the proof effort. Indeed the microprogrammer's view of the control structure is checked with the actual structure of the microprogram.

The system also has the additional advantages of assembler level systems.

- The low level code is translated into a high level program in a modelling language with formally defined semantics. This program
  - could be checked by the user, giving an early opportunity to detect errors, and
  - could be used as the basis of a fast simulator.
- Also, the tools can be used by microprogrammers to check their understanding of individual microinstructions.

Unlike the assembler level systems, the translation is performed automatically from a formal description of the microarchitecture, rather than by a specially written translation program.

Advantages of the high level systems are also obtained.

- The microprogram is treated as an actual program, rather than just as data to an interpreter. This allows the proof effort to be naturally divided according to the structure of the program.
- The program that the software verification techniques are applied to is hierarchical and modular. It does not have the tangled structure, liable to be present in the actual microcode. This simplifies the verification task. However, since the microcode does not have to be written this way, efficiency is not lost.
- The system forces the user to provide formal documentation of the microprogram, rather than just requiring a functional specification.
- The verification process can be incorporated as an integral part of program development.

Chapter 3

The SPOOK Language

This chapter describes the syntax and semantics of the SPOOK language. It is used to describe both the host and skeleton and as the modelling language for the high level microprogram produced by the system. It is a simple imperative language, the expressions of which are based on those of BSPL. The main differences from BSPL expressions are that an extra signal, U (completely undefined), is used; several different kinds of data object are introduced—state words model the values in state storing devices and history and time objects model input-output; the semantics of the operators is specified more completely, and some new operators such as subscription update and merge have been added. The command set consists mainly of standard imperative commands, the semantics of which have been extended to deal with word rather than boolean conditions. New commands include MI and CALL_MODULE which are used within the skeleton and BLOCK, the semantics of which is designed for describing host architectures. Assertions in the unquantified predicate calculus are also supported. A knowledge of the language is a prerequisite for the remainder of this dissertation. However, the details of the individual operators and commands can be skimmed on first reading and referred to later as necessary.

3.1 Introduction

The SPOOK language is based on the hardware description language, BSPL [Richards 86]. It is used for three purposes within the system:

- describing the host program,
- describing the skeleton, and
- as the modeling language for the high level microprogram.

The use of a single language for these three tasks simplifies the production of the high level microprogram. It can be produced using simple rewriting techniques on the host and skeleton. The host and skeleton have different needs, however, and use different subsets of the language.

3.1.1 The Needs of the Host

The host program requires language constructs suitable for modelling hardware in a concise fashion. It also needs to have a fairly simple structure and semantics to aid the formation of collapsed host programs. Furthermore, it is advantageous if no annotations are required in the high level microprogram at points within the collapsed host programs. This would allow the annotations to be placed within the skeleton prior to the verification attempt rather than it having to be done after the high level program had been formed. It is therefore desirable that the host program contain no branches or loops. In order to make the host program as concise as possible, and ensure it has the desired structure for the model used, a number of sugaring features are provided. The user writes an abbreviated version which is expanded into the pure SPOOK language by the system. This abbreviated form is described in Chapter 4.

3.1.2 The Needs of the Skeleton

The purpose of the skeleton is to provide structure for the high level microprogram. Suitable high level constructs for providing this structure and modularising the code are required. Also, a means of indicating the positions of, and states represented by, the nodes of the state transition graph of the microprogram are needed. Furthermore, provision must be made for placing assertions within the skeleton.

3.2 SPOOK Programs

A SPOOK program consists of a series of declarations of variables followed by definitions of the skeleton, host program, etc. Each definition may be read in to the system separately, provided that the appropriate declarations are given with each.

The definitions may define

- the **host program**,
- the **skeleton**,
- **modules** and
- **predicates**.

Any number of modules and predicate definitions, but only one host and one skeleton, may be given. Each is introduced using an appropriate keyword followed by the body.

3.2.1 The Host Program

The host program describes the underlying microarchitecture of the machine being verified. It is introduced using the keyword `HOST_PROGRAM` followed by the command.

`HOST_PROGRAM` C

The use of the host program is dealt with in detail in Chapter 4.

3.2.2 The Skeleton

The skeleton gives a high level description of the state transition diagram of the microprogram and also incorporates the target specification in the form of a partial correctness specification. It is introduced using the keyword `SKELETON`.

`SKELETON` $\{P\}\ C\ \{Q\}$

The partial correctness specification consists of the precondition (P), a command (C) and a postcondition (Q). The skeleton is dealt with in Chapter 5.

3.2.3 Module Definitions

Modules provide an abstraction mechanism for breaking the skeleton into intellectually manageable pieces. Each module definition is introduced with the keyword `MODULE`. The body of the module consists of a partial correctness specification. A name (n) identifies the particular module.

`MODULE` $n\ \{P\}\ C\ \{Q\}$

3.2.4 Predicate Definitions

A mechanism is provided for defining the predicates which occur within assertions. These definitions may then be expanded during the proof process. The following syntax is used:

`|-` n_0 `(` n_1 `;...;` n_k `)` `=` *body*

Each of the n_i are names and *body* is a predicate calculus statement.

3.3 Declarations

All variable names within a SPOOK program are declared at the start of the program. Their scope extends to the whole of the program. The declaration gives the type of the value stored by the variable and also an indication of the use the variable is put to. It is given by a keyword which may be one of the following:

- LOCAL
- STATE_VARIABLE
- INPUT
- OUTPUT
- TIME
- GHOST

The different types of variable will be discussed in more detail in the subsequent sections.

3.3.1 Names

Syntactically a SPOOK variable name is any combination of letters, digits, slashes, primes and underlines, with the proviso that it does not start with a digit. Upper and lower case letters are treated as distinct syntactic tokens. A number of conventions are used with respect to names, though these are not enforced.

- Names which end with a prime are used for locals which represent the lines into a state variable.
- Names consisting of no lower case characters are reserved for operators and predicates.
- Names whose first letter is an upper case letter are used to represent ghosts and module names.

3.3.2 Signals

Signals represent the values occurring on wires and in state storing flip flops. The set of signal values is extended beyond simple high and low voltage values. A signal may be one of five values.

- 0—Binary digit zero
- 1—Binary digit one
- X—Unknown value from 0 or 1 but not Z or U
- Z—High impedance
- U—Completely undefined

The signal set is basically BSPL's, extended to include the U signal. This extension is discussed below.

0,1—Binary digits Zero and One

The binary digits zero and one model the normal low and high voltage values carried by wires. In certain situations they are also interpreted as truth values with 1 interpreted as truth.

X—Undefined

The X value represents a signal which is either high or low, but where the actual value is unknown or unimportant. It is useful for giving partial specifications. For example, in the S-Machine, the microprogram is held in read only memory, though does not fill all locations. The remaining locations contain random values. The specification of the S-Machine uses X values at these addresses. X values are also used in the Orion specification. When a value is popped from the hardware stack, the stack pointer is decremented and the value at the previous top position becomes "undefined". This is modelled using X values.

In general, the operators of SPOOK give results that are as specific as possible when acting on X signals. The X signals are viewed as being 0 or 1. The possible results are then determined and compared. If all possibilities are identical, that is the result. If the result could be 0 or 1, then X signals are returned.

Z—High Impedance

The Z signal is used to model high impedance values. The basic storage device modelled is the flip flop. If a Z value is input, the flip flop's internal state is unchanged. Constructs of the language ensure that in the host, signal wires are given Z values if given no other value so the state will remain unchanged by default. This is useful since in any one clock period, a large proportion of the state will be unaffected. Z signals are also useful when modelling the joining of wires. If one wire holds a high impedance value, the result will be the one on the other wire.

U—Completely Undefined

The semantics of some BSPL constructs is not completely defined, and the U signal has been added to help iron out these idiosyncrasies. It is an extension to the BSPL signal set representing a completely undefined value. It may be a digital one, zero, high impedance, or indicate an error or race condition. It provides a uniform method by which errors may be raised. It is used in various situations.

- It is intended to be returned as a result when expressions have invalid arguments such as when Z values are added. It is therefore used when, according to the BSPL report [Richards 86], BSPL would generate an "error".

- It is also used to represent the "undefined results" of BSPL, such as occur, for example, when a control signal selects a source dependent on its value being 0 or 1, but instead has an X or Z value. The presence of such a result implies that the host is not performing correctly, and any verification attempt should fail. For example, a random microinstruction might be being evaluated. In this context "undefined" does not mean X, since the result could be 0, 1, X, Z or an error, rather than just 0 or 1.

- It is used to represent race conditions being set up due to loops which are not broken by a register. A BSPL specification would recurse infinitely in the presence of such mutually dependent signals. A separate analysis of the code would be required, to prevent it occurring.

- Finally it is used to detect attempts to form information storing devices such as flip flops other than those modelled explicitly. Again, in BSPL, a separate analysis of the step function would be required to ensure such devices were not created.

It is not an error for a signal wire to carry a U value, even though it may have occurred due to one of the forms of failure mentioned above. The U value must be tested for explicitly. This simplifies the host program collapsing process. This will be discussed in more detail in Chapter 4.

3.3.3 Types

The type of a SPOOK object consists of a *kind* and, where appropriate, a *length* giving the number of signals involved.

Kinds

All expressions within SPOOK have an associated *kind* chosen from the following:

- **word**—modelling the values on bundles of wires in the combinatorial logic,

- **state word**—modelling the values in flip flops and registers,

- **history**—modelling traces of values input and output, and

- **time**—modelling values of time.

The kinds of variables may be deduced from the declarations as given by Figure 3.1.

Lengths

Objects of the various kinds, apart from Time objects, are made from bundles of signals of some fixed size. Objects of these kinds have an associated *length* giving the number of signals involved.

Keyword	Kind
`LOCALS`	word
`STATE_VARIABLES`	state word
`INPUT`	history
`OUTPUT`	history
`TIME`	time
`GHOST`	word

Figure 3.1: The *kinds* associated with keywords in declarations

Type Checking

The type of an expression is the combination of its kind and, where appropriate, its length. The kinds and lengths of all variables must declared in SPOOK. Operators are strongly typed with respect to kinds. The kinds of operands and results of all operators is fixed and given by kind rules so kind checking can be performed for all expressions. The operators involving lengths are overloaded with respect to those lengths. Each corresponds to a range of operators depending on the operand lengths. The lengths of all expressions can, however, be deduced statically from length rules. These give relations which must hold between the lengths of operands and results. Since all atomic expressions—names and constants—have fixed lengths, a depth first algorithm can be used to scan expressions. The lengths of the operands are deduced and it is checked that they obey the length rules. The length of the result can then be determined from the length rules. The pure SPOOK language does not allow decimals or operators which truncate or pad the operands, so the lengths of operands can be deduced statically. These features appear only as sugaring, described in Chapter 4. The problems of type checking such sugared expressions are dealt with in that chapter.

3.3.4 Declarations

Declarations of time variables which store time values consist of the keyword `TIME` followed by a list of names.

TIME n_1 $n_2 \ldots n_k$

The declaration,

```
TIME
  Start_Time
  time
```

declares the names `Start_Time` and `time` to be time variables and have kind, time. Declarations of all other variables have the form

KEYWORD n_1: l_1 n_2: $l_2 \ldots n_k$: l_k

where each l_i gives the length of the associated name n_i. Lengths may be given in one of two forms: either a single integer, or a pair of integers using the syntax *i* -> *i*. In the former case, the length is the given integer. The declaration

```
STATE_VARIABLES
  pc : 16
```

declares name pc to be a state variable of length 16 and kind, state word. The second form gives an abbreviation intended to be used for variables representing memory structures. The first integer gives the number of bits in the address, and the second gives the number of bits in each memory word. The declaration,

```
STATE_VARIABLES
  mem : 13 -> 16
```

would be used to describe mem as a state variable of length $2^{13} \times 16$ representing a memory consisting of 2^{13} memory words of length 16. The additional information is not used by SPOOK. It is present simply to make the declaration more understandable. The above is identical to a declaration

```
mem : 131072
```

3.4 Word Expressions

Signal wires are grouped into bundles which are used to connect combinatorial logic. The kind of the values carried in such bundles is the word. A word value consists of a sequence of signals from the full signal set {0, 1, X, Z, U}.

The majority of operators return results of kind word, and require the operands to be of that kind. The SPOOK word operators are based on those of BSPL, though, unlike in BSPL, the semantics of SPOOK operators are completely defined. Some new operators have also been added. The semantics of the BSPL based ones have been extended to include the U signal, formalising the results described as "an error" and "undefined" in the BSPL definition. They are designed to model functions implemented by combinatorial logic. To keep the semantics simple, and ease verification, all operators are purely functional, having no side effects. In addition to the operators described in this section, two others return word results: GET and SW. These take non-word arguments and so are described in later sections.

3.4.1 Word Constants

Word constants are prefixed by #b. They consist of a sequence of signal values.

#b$s_1 s_2 \ldots s_k$

Here, each of the s_i is a signal from the full set {0, 1, X, Z, U}. The length of the constant is the number of actual signals. The signals may be arbitrarily interspersed with underline characters which have no semantic significance, and do not contribute to the length of the constant. The constant,

```
#b_01X_1ZU_0
```

represents a word of length seven.

3.4.2 Local Variables

Within the host program, word values are stored in local variables. These give names to wires within the combinatorial logic. Locals are declared using the keyword LOCALS. They have an associated length which is the number of wires in the bundle that they name.

3.4.3 Ghost Variables

In the skeleton, word values are stored in ghost variables. By convention, the names of ghosts start with an uppercase letter. The use of ghost (or auxiliary) variables is common practice within software verification where they are used to fix the initial values of variables within the precondition. They have a similar use in the SPOOK system, storing the values of the registers and memory of the device at the start of the microprogram. For example, a precondition might be

```
{ Pc = SW pc }
```

asserting that the ghost Pc holds the initial value of register pc (converted to a word value by the SW cast operator). The following postcondition then asserts that register pc should be incremented.

```
{ SW pc = Pc + #b_0000_0001 }
```

Ghosts can also help make the skeleton and assertions more readable. For example, in the skeleton and annotations of Gordon's computer, the expression giving the opcode of the instruction being executed is abbreviated using a ghost Op. This means that throughout the skeleton and assertions Op can be used instead of the more complex expression below.

```
(Mem !16 Pc)[15..13]
```

This syntax is described later in this chapter. Ghost variables are declared using the keyword GHOSTS, and are given an explicit length.

3.4.4 Concatenation

The concatenation operator provides a means for combining signal words to make longer ones. Concatenation expressions have the syntax

w_1 , w_2

where w_1 and w_2 are words. The two operand words are concatenated end to end, with the second at the least significant end of the result. Both operands and results have kind word. The length of the result is the sum of the lengths of the arguments. The following expressions are equivalent

#b01U , #bX1X $\equiv$ #b01U_X1X

3.4.5 Selection

The selection operator provides a means by which individual signal wires may be selected from a signal word. A select expression has the form

w [s_1, s_2,... ,s_k]

The wires selected are indicated by the s_i which give their positions, numbered from the least significant (rightmost) end, starting at zero. Each s_i may be a single integer, selecting a single wire position, or a pair of integers in the form $i_1 \mathinner{.\,.} i_2$ giving a range of positions. The latter form is an abbreviation for a sequence of single integer positions. If a range is given with the smaller value first, the effect is to reverse the order of the signal positions in the range selected. All positions are given as integer constants so that the wire referred to is known at compile time. It is a compile time error for a selected position to be greater than or equal to the length of the word argument. If any signal in the operand word is a U signal, the result word consists of U signals, even if no U signal position is selected. This prevents U signals and thus errors from being lost.

The operand and result have word kind. The length of the result is the number of wires selected. For example, the following expression forms a word of length 4, consisting of the 25th, 24th, 23rd and 20th signals of `instr` concatenated together, with the 20th becoming the least significant signal.

```
instr[25..23,20]
```

The following are also equivalences:

`#b_01X_101X[0,2..3]` ≡ `#b_X01`

`#b_0U1[2,0]` ≡ `#b_UU`

3.4.6 Subscription

The subscription operator allows wires within a word to be selected in a fashion modelling the access of memory. A subscription expression has the following form.

w_1 !n w_2

The first argument (w_1) gives the memory word to be accessed. An address word, which need not be a constant, is then given as a second argument (w_2). An explicit integer (n) gives the size of the words accessed. If the first operand contains U signals or the second contains Z or U signals, the result is a word where each constituent signal is U. A word of U signals is also returned if the addressed position is out of range. If the address contains X signals, then all the possible results, assuming the X's were interpreted as 0's or 1's, are determined. These are compared and the most accurate result possible returned. The comparison is made by the SHAKE meta-operator (which is not available as a SPOOK operator in its own right). The shake of two words is defined in a signalwise fashion as given below.

- If the signals are equal, that is the result signal,
- otherwise, if either is U or Z, U is the result signal,
- otherwise X is the result signal.

The arguments and result have kind word. The size of the result, and thus that which the memory is partitioned into, is given by the explicit integer, *n*.

The expression,

```
rom !29 mpc
```

selects the `mpc`-th (where `mpc` is interpreted as an integer) group of 29 signals, counting from the least significant end, of `rom`. The following example illustrates the effect of an illegal address:

```
m !3 #b0Z ≡ #bUUU
```

3.4.7 Logical Operators

SPOOK provides the standard logical operators.

- And (w_1 `&` w_2)
- Or (w_1 `|` w_2)
- Equivalence (w_1 `EQV` w_2)
- Non-equivalence (w_1 `NEQV` w_2)
- Not (`.~` w)

These all work in a signal-wise fashion applying a function to each corresponding pair of signals independently. The signal functions semantics are given by Figure 3.2. They are identical to those for BSPL, except that U signals are used where BSPL returns an "error" and when the arguments contain U signals. The kind of result and arguments is the word. These operators take equal length operands and give a result of the same length.

The following equivalences hold.

```
.~#b_X01 ≡ #b_X10

#b_01X_001 & #b_101_X01 ≡ #b_00X_001

#b_01X_001 | #b_101_X01 ≡ #b_111_X01

#b_01X_001 EQV #b_101_X01 ≡ #b_00X_X11

#b_01X_001 NEQV #b_101_X01 ≡ #b_11X_X00
```

.~	0	1	X	Z	U
	1	0	X	U	U

		S₂				
	&	0	1	X	Z	U
	0	0	0	0	U	U
	1	0	1	X	U	U
S_1	X	0	X	X	U	U
	Z	U	U	U	U	U
	U	U	U	U	U	U

		S_2				
	|	0	1	X	Z	U
	0	0	1	X	U	U
	1	1	1	1	U	U
S_1	X	X	1	X	U	U
	Z	U	U	U	U	U
	U	U	U	U	U	U

		S_2				
	EQV	0	1	X	Z	U
	0	1	0	X	U	U
	1	0	1	X	U	U
S_1	X	X	X	X	U	U
	Z	U	U	U	U	U
	U	U	U	U	U	U

		S_2				
	NEQV	0	1	X	Z	U
	0	0	1	X	U	U
	1	1	0	X	U	U
S_1	X	X	X	X	U	U
	Z	U	U	U	U	U
	U	U	U	U	U	U

Figure 3.2: The semantics of the logical operators

		S_2				
	*	0	1	X	Z	U
	0	U	U	U	0	U
	1	U	U	U	1	U
S_1	X	U	U	U	X	U
	Z	0	1	X	Z	U
	U	U	U	U	U	U

Figure 3.3: The semantics of join

3.4.8 The Join Operator

The join operator is used to model the joining of wires. A join expression has the form

$w_1 * w_2$

It works in a signalwise fashion as indicated in Figure 3.3. If one signal wire holds a high impedance value, then the corresponding signal from the other becomes the result in that position. Otherwise, a U value is returned in that position. Both arguments and results are of kind word. The operands are the same length and that will be the length of the result. The following example illustrates the join operator.

```
#bZZZ * #bX10 ≡ #bX10
```

3.4.9 The Arithmetic Operators

Three arithmetic operators are provided.

- Addition (w_1 + w_2)
- Subtraction (w_1 - w_2)
- Monadic minus (- w)

The arithmetic add operator implements unsigned word addition with no carry in. If the operands contain only binary signals, then the result will be the exact unsigned binary sum. If either operand contains X then the result will contain at least one X but will be as specific as possible. As with other operators, the possible results are determined and then combined using the SHAKE operator. If either operand contains U or Z signals, then the result will be a word of appropriate length, containing only U signals.

The subtraction operator performs two's complement subtraction and is defined in terms of the addition operator.

$$w_1 - w_2 \overset{\text{def}}{\equiv} (w_1 + .\tilde{\ } w_2 + 1)\texttt{[}n\texttt{..0]}$$

Here n represents the length of the first argument and 1 represents a word of the appropriate length consisting of zeros, except for the least significant position which is digital one.

The monadic minus is similarly defined in terms of the subtraction operator.

$$-w \overset{\text{def}}{\equiv} 0 - w$$

Here 0 represents a word having the same length as e, consisting solely of zero signals.

For each of the arithmetic operators the kind of result and arguments is the word. The result is of length one greater than the arguments. The following are equivalences:

```
#b01X_001 + #b101_X01 ≡ #bX_XXX_X10
#b01X_001 - #b101_X01 ≡ #b0_XXX_X00
- #b01 ≡ #b0_11
```

3.4.10 The Shift Operators

Both left and right shift operators are provided.

- Left shift (w_1 >> w_2)
- Right shift (w_1 << w_2)

The shift operators differ from their BSPL counterparts in that the amount shifted is not a fixed value. Instead it may be given in the form of an arbitrary signal word as given by the second argument. This word is interpreted as an integer. This is of more use, as illustrated in the BSPL machine code level specification for the S-Machine. In the BSPL report [Richards 86] this specification contains a syntax error, since the shift operators are given arbitrary second operands. In BSPL this would also not be well-typed, since the result type depends on the value of this second argument. In SPOOK the kind of both arguments and the result is the word. The lengths of the result and the word to be shifted are the same. When shifting left, the extra signals shifted out are lost. When shifting right, extra zero signals are added onto the most significant end. The effect of the BSPL semantics can still be achieved using concatenation and the selection operator.

$w >>_{BSPL} 1 \equiv (w$ >> #b_1) [n..0]

$w <<_{BSPL} 1 \equiv$ (#b_0, w) << #b_1

In the first equivalence n represents an integer one less than the length of the first operand.

For both operators, if either operand holds U signals, or the second operand holds Z signals, then the result is a word of the appropriate type consisting of U values. If the second operand holds X signals then the result is as accurate as possible. As with the subscription operator, the X's are interpreted as 0's and 1's, and the possible results determined. The most accurate result is then determined using the SHAKE operator described earlier. For example, the following equivalences hold.

#b011 >> #b0X $\equiv$ #b0X1

#b01X << #b10 $\equiv$ #bX00

3.4.11 The Relational Operators

Several operators are provided for comparing words.

- Equal (w_1 .= w_2)
- Not equal (w_1 .~= w_2)
- Less than (w_1 .< w_2)
- Greater than (w_1 .> w_2)
- Less than or equal (w_1 .<= w_2)
- Greater than or equal (w_1 .>= w_2)

The comparisons are only made up to binary digits. The result is not necessarily a binary digit and so cannot be strictly interpreted as a truth value. If X signals occur in the operands then the results may contain X signals. If Z or U values occur the result will be a U valued word. The equality operator, for example, does not test for the actual equivalence of words. The syntax is slightly different to BSPL in that each operator starts with a period. The equality operator, for example, is ".=" rather than "=". This is to avoid confusion with the equality used in assertions, which gives true equality. For example,

#bU = #bU $\equiv$ T

whereas

```
#bU .= #bU ≡ #bU
```

The semantics are intended to be the same as for BSPL, though the U signal is used to clarify the meaning of applying these operators to Z signals. The kind of result and arguments is again the word. Each operator returns a result of length 1. The operands should be of the same length.

Equals

The equals operator returns a result indicating whether its operands are the same (if binary). If either contains a U or Z signal, the result is #bU. If they are binary (containing only 0's and 1's) and identical, the result is #b1. If they are binary but different, the result is #b0. The only remaining case is when one or both operands contains an X signal. Here, all possible results which could occur if the X signals were interpreted as 0's and 1's are determined, and the SHAKE of the results returned. This means that if there is a definite difference (one operand has a 0 where the other has a 1), #b0 is returned. If not #bX is the result. For example,

```
#b01X1 .= #b01X0 ≡ #b0
#b01X1 .= #b01X1 ≡ #bX
```

Not Equals

The not equals operator is defined in terms of not and equals.

$$w_1 \;\texttt{.\~{}=}\; w_2 \overset{\text{def}}{\equiv} \texttt{.\~{}}\;(w_1 \;\texttt{.=}\; w_2)$$

For example,

```
#b01X1 .~= #b01X0 ≡ #b1
```

Less Than

The less than operator returns a result which indicates whether the first operand, if interpreted as an unsigned integer, is less than the second when similarly interpreted. If either operand contains U or Z signals, the result is #bU. If they are both binary and the first is less than the second when interpreted as an integer, then #b1 is returned. If they are binary but the first is the greater or they are equal, the result is #b0. In the remaining case, at least one operand will contain X signals. As with the equals operator all possible results which could occur if the X signals were interpreted as 0's and 1's are determined, and the SHAKE of the results returned. For example,

```
#b01X1 .< #b01X0 ≡ #bX
#b00X1 .< #b01X0 ≡ #b1
```

Greater Than

The greater than operator is defined in terms of less than.

$$w_1 \texttt{ .> } w_2 \overset{\text{def}}{\equiv} w_2 \texttt{ .< } w_1$$

For example,

$$\texttt{\#b00X1 .> \#b01X0} \equiv \texttt{\#b0}$$

Less Than or Equal

The less than or equal operator returns a result which indicates whether the first operand, if interpreted as an unsigned integer, is less than or equal to the second when similarly interpreted. If either operand contains U or Z signals, the result is **#bU**. If they are both binary and the first is less than or equal to the second when interpreted as an integer, then **#b1** is returned. If they are binary but the first is the greater, the result is **#b0**. In the remaining case, at least one operand will contain X signals. Again, all possible results which could occur if the X signals were interpreted as 0's and 1's are determined, and the SHAKE of the results returned. For example,

$$\texttt{\#b001 .<= \#b1X0} \equiv \texttt{\#b1}$$

It should be noted that

$$w_1 \texttt{ .<= } w_2$$

is not equivalent to

$$(w_1 \texttt{ .< } w_2) \texttt{ | } (w_1 \texttt{ .= } w_2)$$

since, for example,

$$\texttt{\#b0 .< \#bX} \equiv \texttt{\#bX}\text{, and}$$

$$\texttt{\#b0 .= \#bX} \equiv \texttt{\#bX}\text{, so}$$

$$\texttt{(\#b0 .< \#bX) | (\#b0 .= \#bX)} \equiv \texttt{\#bX}\text{, though}$$

$$\texttt{\#b0 .<= \#bX} \equiv \texttt{\#b1}$$

Greater Than or Equal

The greater than or equal to operator is defined in terms of the less than or equal to operator.

$$w_1 \texttt{ .>= } w_2 \overset{\text{def}}{\equiv} w_2 \texttt{ .<= } w_1$$

For example,

$$\texttt{\#b001 .>= \#b1X0} \equiv \texttt{\#b0}$$

3.4.12 Well-definedness

The syntax of well-definedness expressions is

```
WD w
```

The WD operator checks for the well-definedness of its operand. That is, it returns `#b0` if its operand contains a U signal and `#b1` otherwise. This is a situation where the digital one and zero values are interpreted as truth and falsity, respectively. `WD` is used in the host program to trap the occurrence of errors, indicated by the prescence of U values. It is not used to model actual hardware. The kind of both the argument and result is the word. The result is always of length 1.

The following are equivalences.

```
WD #b00U1 ≡ #b0
WD #bZX ≡ #b1
```

As with other SPOOK operators, WD has a very concise name. This is because often very large expressions are generated during the proof process. These are more readable if the common operators are concise.

3.4.13 Z_WORD

`Z_WORD` expressions have the form

```
Z_WORD n
```

where n is an explicit integer. It returns a word consisting of n Z signals. It is useful on occasions when large Z valued words are required (which can be often). This situation arises when a variable representing a memory is to be assigned Z values. In this case it would be impractical to use the actual constant.

3.4.14 Conditional Expressions

Two conditional operators are provided: `GUARDED` and `SWITCH`. The former is a general purpose, multiple branch conditional, whereas the latter provides a multiple branch case facility using word labels.

GUARDED

Guarded expressions have the following form:

```
GUARDED
 || g1 => w1
 || g2 => w2
      .
      .
 || gn => wn
 DEFAULT wn+1
```

The guarded operator takes a set of guards (g_i) and corresponding result operands (w_i). An extra result operand (w_{n+1}) is also taken as a default. The operands and result are all of kind word. The length of the result is the same as the length of each of the result expressions. The guards are all of length 1. The result depends on the value of all the guards.

- If any guard is `#bU`, `#bZ` or `#bX`, or if more than one guard is `#b1`, then a word of U signals is returned.
- If all guards are `#b0`, then the default result is returned.
- Otherwise, the result corresponding to the `#b1` guard is returned.

Because U values are returned rather than errors being generated, a guarded expression can be evaluated strictly. The conditionals are among the few operators which do not necessarily propagate U values from the operands to the result.

For example,

```
GUARDED
 || res_is_0 => #b000
 || res_is_U => #bUUU
 DEFAULT #bZZZ
```

evaluates to `#b000` when `res_is_0` is `#b1` and `res_is_U` is `#b0`.

SWITCH

Switch expressions have the following form:

```
SWITCH w
 || g1 => w1
 || g2 => w2
      .
      .
 || gn => wn
 DEFAULT wn+1
```

The switch operator takes a subject operand, and a set of distinct, binary (that is, consisting only of 0 and 1 signals) word labels with corresponding result operands. As for guarded expressions a default result is also taken. The chosen result depends on the subject.

- If the subject contains U or Z signals, the result is a word of U signals.
- Otherwise, if the result is equal to one of the labels, the corresponding result operand is returned.
- If the subject is binary, but equal to non of the labels, the default result is returned.

- Otherwise, the subject must contain at least one X signal.
  - It is first determined which labels the subject would match, if the X signals were treated as 0 or 1.
  - The SHAKE (defined previously) of the corresponding results is returned.

The operands and result of the switch operator are all of kind word. The length of the result is the same as the length of each of the result expressions. The length of the subject and the labels is the same. For example,

```
SWITCH alu_fun
 || #b00 => #b111
 || #b01 => #b010
 DEFAULT #bZZZ
```

evaluates to #bX1X when alu_fun is #b0X.

Discussion

The conditional operators replace the guarded rules of BSPL. In BSPL, if two or more guards in rules with the same left hand side are active, the corresponding results are joined implicitly using the * operator. This simulates the effect of wires taking their value from multiple sources, though complicates the semantics greatly, hindering the collapsing of host programs and proof of verification conditions. Since, on the whole, wires taking their values from multiple sources would indicate an error, this feature was dropped from SPOOK. Instead U valued words are returned. The effect can still be obtained by using the join operator explicitly.

In BSPL, guarded rules for a particular left hand side may be distributed throughout the step function. The conditional expressions, in effect, force them to be placed together. In general this is more readable, since the actual value taken depends on all the guards.

The switch operator is provided in addition to the guarded operator as it provides clean syntax and also eases a problem encountered due to X signals. The guarded operator returns a U signaled word if any of the guards evaluate to #bX since this means that the guard may or may not be active. Thus, more than one guard could be active. With switch, if the subject contains X values, only one of the labels would be matched however the X's were interpreted. There is no possibility of results clashing. The switch can therefore give a non U result, returning X or U values when there is a disagreement in the possible result values. This means that a switch expression can be guaranteed not to return U values much more easily than a guarded expression. It does not depend on ensuring that subexpressions do not return X signals, which in general cannot be done. This is very important when collapsing the host program. Also, switch expressions will simplify more easily since, for example, if all the possible results are the same, that will be the result returned.

Conditional operators are provided, in addition to the similar branching commands described later, for various reasons.

- It provides a neater syntax, avoiding the duplicity of having multiple assignments to the same variable.
- It simplifies the structure of the host program which
  - simplifies the collapsing process, and
  - avoids the need for annotations within the host program.
- It also avoids a combinatorial explosion in the number of verification conditions generated, since each separate guard is not treated as a separate path through the code. Rewriting of the expression may then avoid the need to look at each case separately when proving the verification conditions.

3.4.15 Subscription Update

Subscription update expressions have the form

w_1 `!=` w_2

The result is a word consisting of Z signals except the block of signals at the address indicated by the first argument (w_1), which is a copy of the second argument (w_2). The address is determined by interpreting w_1 as an integer and multiplying this by the length of w_2. If the first argument contains X, Z or U signals, or the second argument contains U signals, the result is a word of U signals. Both operands and results have kind word. The length of the result is the result of multiplying the largest possible address by the length of the second argument. For example, the following equivalence holds

`#b_10 != #b_11` $\equiv$ `#b_ZZ_11_ZZ_ZZ`

The subscription update operator is used when altering a part of a state variables value at a given address within it, such as altering one memory word within a state variable representing memory. In BSPL this would be done using a rule of the form

mem `!16` *a* `:=` *v*

In SPOOK it would be implemented by an assignment of the form

mem `==` *a* `!=` *v*

Although the syntax is more obscure, this simplifies the semantics of the language, since only simple assignments occur.

3.5 State Word Expressions

The fundamental state storing device modelled in the SPOOK system is the flip flop. These notionally have an input signal wire and output signal wire. They possess one bit of internal state which may store a signal from the set {`0`, `1`, `X`}.

Input Value	Next State Q_{t+1}
0	0
1	1
X	X
Z	Q_t
U	X

Figure 3.4: The behaviour of flip flops

All flip flops are controlled by a single universal clock. On a clock tick they change state simultaneously according to their current state and input value, outputting the new state. Their behaviour is given in the table of Figure 3.4. According to this table, the value X is stored if a U signal is input to the flip flop. This is given for completeness. In practice U signals should never be input to the state variable in this way, but would be trapped earlier. For all values of t, Q_t has one of the values from {0, 1, X}. If a Z value is input to a flip flop, its state is unchanged. Two operators, **WS** and **MERGE**, are provided for converting words into state words.

3.5.1 State Word Constants

Flip flops are grouped together into registers. These hold state word values. They are similar to words except that the constituent signal values are limited to the set {0, 1, X}. State word constants are prefixed by **#s** and consist of a sequence of signals, s_i from the above set.

#s$s_1 s_2 \ldots s_k$

The constant **#s_010X** is a legal state word, whereas **#s_01ZX** is not. As with words, the signals may be arbitrarily interspersed with underline characters.

3.5.2 State Variables

Registers are represented by state variables. These store state word values. The total state of a machine is held in the set of state variables. State variables are declared using the keyword **STATE_VARIABLES**. They have an associated type which is the length of the state word that they store.

3.5.3 Word to State Word Conversion

Word to state word conversion is performed using the **WS** cast operator. **WS** expressions have the form

WS w

		S_2				
	MERGE	0	1	X	Z	U
	0	0	1	X	0	X
S_1	1	0	1	X	1	X
	X	0	1	X	X	X

Figure 3.5: The semantics of the merge operator

Given a word, *w*, it returns the corresponding state word, mapping 0, 1 and X signals onto themselves. U and Z signals are mapped onto an X signal. This should never occur if the operator is used properly. This is ensured by the sugaring techniques described later. The kind of the operand is word and that of the result is state word. Both operand and result have the same length. The following are equivalent.

WS #b10X $\equiv$ #s10X

3.5.4 MERGE

The merge operator models the action of the clock tick on flip flops. A merge expression has the form

MERGE (*s*; *w*)

It takes two arguments: a state word, *s* representing the flip flops' value and a word *w* representing the value on the inputs to the flip flops. Merge acts in a signal-wise fashion as given by the table in Figure 3.5. If a U signal occurs as part of the word argument then an X signal is returned. As with the WS operator, this situation should never occur. If the second argument signal is 0, 1 or X, that is the result signal. If it is Z, the result signal is taken from the first argument.

If used in conjunction with subscription update, a state word can be obtained which has the same value as the original, except at the given address, where the new value is used. The expression

```
MERGE(reg; addr != new_val)
```

would be used to update a register `reg` at address `addr` with new value `new_val`. When `reg` has value `#s_00_00_00_00`, `addr` has value `#b_10` and `new_val` has value `#b_11`, this expression would be

```
MERGE (#s_00_00_00_00; #b_10 != #b_11)
```

which is equivalent to

```
MERGE (#s_00_00_00_00; #b_ZZ_11_ZZ_ZZ)
```

which in turn is

```
#s_00_11_00_00
```

3.5.5 State Word to Word Conversion

`SW` converts state words back into words. `SW` expressions have the form

```
SW s
```

Given a state word, s, it returns the corresponding word, mapping 0, 1 and X signals onto themselves. The kind of the operand is state word and that of the result, word. Both have the same length. The following is an equivalence.

```
SW #s01X ≡ #b01X
```

3.6 Time Expressions

Time expressions are used to model time, which is required for input and output. A discrete model of time is used. The integers could be used for this purpose though in SPOOK an abstract type is used, providing just increment (`INC`) and decrement (`DEC`) operators. This simplifies theorem proving, avoiding the need for integer arithmetic, and is all that is required for the examples considered. In general, all time values are relative to some start time such as the start of a macro cycle, so time constants are not required. In the examples considered, the offset from the start time is always fairly small. For more complex examples it could be larger, and so constants and operators such as addition would be useful. For example, to refer to the time 30 time units after `Start_Time`,

```
Start_Time + 30
```

could be used, rather than the voluminous

```
INC(INC...(INC Start_Time))...)
```

3.6.1 Time Variables

Time variables store time values. The keyword `TIME` is used to declare them. The variable `time` is a distinguished time variable which represents the current time. It must also be declared. Other time variables are used as ghosts.

3.6.2 INC

The `INC` operator increments time. An inc expression has the form

```
INC t
```

It takes a time argument (t) and returns a time result, representing a time instance later.

3.6.3 DEC

The DEC operator decrements time. A dec expression has the form

```
DEC t
```

It takes a time argument (t) and returns a time result representing a time instance earlier.

3.7 History Expressions

History words are values which are effectively infinite arrays indexed by time. The value at a given time location is a word giving the value on an input or output port to and from the external environment at that time. After each clock tick new input values are presented on the input ports and the combinatorial logic propagates new values to the output ports. The length associated with a history word is the length of the words which it stores. For a particular value, all constituent words should have the same length. The actual history values have no constant representation, since they have infinite size. They are represented abstractly using the operators GET and PUT. These allow values at particular times to be accessed and set. Only the PUT operator returns a history value.

3.7.1 Input and Output Variables

Input and output ports are modelled by input and output variables, respectively. In BSPL bidirectional *inout* variables were also provided. These have not been included in the present version of SPOOK, since they were not required for the examples considered. Inout variables could easily be included in later versions of the language. The same effect could be simulated using individual input and output variables to represent the inout port.

Input variables can only appear in a right hand context. There is no mechanism for changing their value, since this is done by the external environment. Input and output variables cannot hold or return U values. They are declared using the keywords INPUT and OUTPUT.

Currently no facilities are provided for history ghost assignment within the skeleton. This might be useful for history ghost variables to keep a record of the expected values of outputs. Virtual programming in this way was suggested by Clint [Clint 73] [Clint 84].

3.7.2 PUT

PUT updates a history value at a particular time, and is used for output. A put expression has the form

```
PUT(h; t; w)
```

It takes a history value (h), a time (t) at which the value is to be updated and the new value (w) to be placed there. It returns a new history value. The only other type of expression which can return a history value is a history variable.

3.7.3 GET

GET is used to access a word from a history value at a particular time, thus modelling input. This is the only way that history values may be inspected. A get expression has the form

GET(h; t)

It takes a history operand (h) and a time operand (t) returning a word.

3.8 Assertions

Assertions give conditions which should hold at the particular point in the program—either as annotations, preconditions or postconditions. They provide formal documentation of the code and are not executable. Assertions are formed by placing curly brackets around predicate calculus statements. An assertion would have the form

$\{ P \}$

where P is a predicate calculus statement. Null assertions are also allowed, consisting of just curly brackets. These are useful when developing programs.

Predicate calculus statements are build up from atomic statements using the standard logical connectives. The allowed atomic statements are

- truth (T)
- falsity (F)
- predicate symbols applied to sequences of expressions

  $$(\textit{pred\_name}\ (e_1;\ e_2; \ldots;\ e_k))$$

- equality between expressions ($e_1 = e_2$)

By convention predicate names consist of upper case characters. Predicates may take arguments that are any expression kind. As with expressions, the built in ones may be notionally overloaded with respect to the length of the arguments. For example,

BINARYP (w)

is a predicate which is true if its word argument consists only of 0 and 1 signals and false otherwise.

The allowed logical connectives are

- Conjunction (/\)
- Disjunction (\/)
- Implication (==>)
- Negation (~)

Statements may also be bracketed. Quantifiers are not currently supported. Where needed, suitably defined predicates may be used instead.

3.9 Commands

SPOOK provides a small set of commands with fairly simple semantics to aid the verification process. The semantics are described formally using an axiomatic semantics given in Appendix A. Several commands such as the block and parallel assignment commands are provided specifically for describing the host, and this is reflected in their semantics. Structuring commands such as `IF` and `WHILE` are provided to allow the skeleton to be structured and a simple module facility is provided to allow it to be hierarchical. No jump commands are provided. A command for indicating the positions where microinstructions are executed within the skeleton is also provided.

The main kind used in expressions is the word. Words of length 1 are used in tests rather than booleans reflecting the way hardware uses words. The semantics of the commands is extended to cope with X, Z and U values. Commands may be bracketed using `$(` and `$)`.

3.9.1 SKIP

The skip command has the form

```
SKIP
```

Execution of the skip command does nothing. This is useful for the default cases of branches.

3.9.2 ABORT

The abort command has the form

```
ABORT
```

Execution of the abort command causes a program to fail. Its presence on an executable path will prevent any proof attempt from being successful. It is used as the default case of branches which are intended to be unexecutable. If they are executable, this will be discovered during verification.

3.9.3 Sequencing

Commands may be composed sequentially using the sequencing operator.

$C_1;\ C_2$

Initially the first command (C_1) is executed, and then the second (C_2). An assertion (R) may optionally be placed between the commands. Such an assertion is not executable but merely provides documentation and aids the proof process.

$C_1;\{\mathrm{R}\}\ C_2$

3.9.4 CHECK

The check command is used to perform dynamic error checks. It has the form

CHECK w

taking a word expression (w) of length 1 as argument. If this expression evaluates to `#b1`, the command does nothing—it is equivalent to a skip command. If the expression evaluates to anything else, the command causes execution to fail—it is equivalent to `ABORT`. The command is similar to a EUCLID legality assertion, which EUCLID compilers must insert to detect run time errors [Wortman 79]. Similar commands are also found in the SPADE system. The major uses of the check command in the SPOOK system are

- to ensure that information used to collapse host programs is actually true at run time when that microinstruction is executed, and
- to detect when U values have been generated in the combinatorial logic of a host program.

The following examples illustrate these uses respectively.

```
CHECK (SW mpc .= #b00000)
CHECK (WD bus)
```

3.9.5 MI

Microinstruction execution commands have the form

MI w

where w is a word expression of length 1.

The `MI` command is used to indicate the points in the skeleton where microinstructions are intended to be executed. It take the place of the node in a state transition diagram. A word condition is provided as an argument. This specifies which microinstruction is being executed, giving restrictions on the state before execution. It might, for example, give the value held in an instruction register. If the condition is not true at run time, the program will abort. Otherwise the effect will be the same as when executing a copy of the host program, i.e., one tick of the microclock occurs. For example, an MI command representing the microinstruction held at location zero in a ROM addressed by a microprogram counter mpc might be:

```
MI(SW mpc .= #b0000)
```

3.9.6 Assignment

Simple assignments have the syntax

name == *e*

where *e* is a word, history or time expression. The kind and length of the expression assigned are the same as the variable that the assignment is made to. Assignments of this form which occur within a host program may only be used to assign to local and output variables. In the following examples the first assigns to a local m and the second to the output ready.

```
m     == rom !29 (SW mpc)
ready == PUT(ready; time; #b0)
```

In the skeleton, assignments may be made to ghost and time variables only. In the following examples, the first assigns to a time variable Start_Time and the second to a ghost Op.

```
Start_Time == time
Op         == Mem !32 Pc
```

3.9.7 Parallel Assignment

Parallel assignments have the form shown below. The kinds of the variables assigned to, and values assigned must be state word or time. The lengths must also correspond where appropriate (that is where the assignment is not a time assignment).

```
PAR
 $(
   v1 :=  e1
   v2 :=  e2
      .
      .
   vn :=  en
 $)
```

The par command is used to model the action of the clock tick in the host program. On the clock tick, all the state variables are updated and time is incremented together. The appropriate assignments are therefore placed in a single par command. The assignment to time is made explicitly for clarity and flexibility. The following is an example of a par command which updates a microprogram counter mpc in addition to incrementing time.

```
PAR
 $(
   mpc  := #s00001
   time := INC time
 $)
```

3.9.8 Blocks

The block command has the form

```
BLOCK C
```

It is designed specifically with the host program in mind and is used to define the extent of a clock tick with regard to the period in which locals retain their values. At the start and end of each block command, all the local variables, as declared at the start of the program, are assigned U values. This ensures that host state cannot be propagated between clock periods using locals. This can only be done using state variables. Furthermore, assignments may not be made to any local outside a block command. Assigning U values to all the locals at the start of the block also allows attempts to create state storing devices and unbroken feedback loops to be detected. This is described in more detail in Chapter 4.

The names of the locals are declared at the start of the program rather than being included explicitly in the block command for the sake of brevity. A large number of locals are liable to be used, and will be identical for each block.

The assignment of the U values is made an implicit part of the semantics to keep the descriptions concise. Also, if the commands were added explicitly, the U constants would be propagated forwards when forming collapsed host programs. This would make the location of errors much harder to identify as the names of variables holding U values would be replaced by U constants.

Examples of the use of block commands are given in Chapter 4.

3.9.9 Calling Modules

As described previously, SPOOK includes a module facility. A module is called using the command CALL_MODULE with a name identifying the module.

```
CALL_MODULE name
```

It provides a macro call facility with no parameter passing. Execution of the call is equivalent to executing the body of the named module. Dynamic binding is used. All variables remain in scope throughout the body of the module. For example,

```
CALL_MODULE Add
```

is equivalent to executing the body of the module named Add.

It is used as a simple way to modularise the skeleton and consequently the proof method. Due to its simplicity, it cannot be used for multiple calls. For that, a proper procedure call facility would be needed, for example when the microcode contained actual procedure calls. The current version of SPOOK does not support full procedure call since CALL_MODULE is sufficient to illustrate the way that the skeleton can be modularised and is all that is needed for the examples considered.

3.9.10 IF Statements

The if command provides a general structured multiple branching facility.

```
IF
  || $b_1$=> $C_1$
  || $b_2$ => $C_2$
        .
        .
  || $b_n$ => $C_n$
  DEFAULT $C_{n+1}$
```

Words of length 1 (b_i) are used as guards, modelling the tests present in hardware. This entails expanding the semantics of traditional branching commands to cope with #bX, #bZ and #bU values. If any guard evaluates to one of these values, abortion occurs. At most one guard should evaluate to #b1 at any time. The corresponding command is executed. The default command is executed if all of the guards evaluate to #b0.

In the following example, if the ghost accumulator Acc is zero the module Jmp is called, if it contains X, Z or U values abortion occurs and otherwise module NoJmp is called.

```
IF
 || Acc .= #b0000_0000 =>  CALL_MODULE Jmp
 DEFAULT                   CALL_MODULE NoJmp
```

3.9.11 Case Statements

The case command provides a case statement in a similar style to the if command.

```
CASE $w$
  || $b_1$ => $C_1$
  || $b_2$ => $C_2$
        .
        .
  || $b_n$ => $C_n$
  DEFAULT $C_{n+1}$
```

A switch is made on a word expression (w). This is compared with each of a set of different binary (consisting of 0 and 1 signals) word constants (b_i). If the word expression evaluates to any of the constants, then the associated command is executed. If it contains any X, Z or U signals then a run time error occurs.

Otherwise the default command is executed. All the constants should have the same kind (word) and length as the switch expression.

In the example below, if ghost, Op, has value #b00, the module Add is executed, if it has value #b01, Sub is executed, and otherwise abortion occurs.

```
CASE Op
 || #b00 => CALL_MODULE Add
 || #b01 => CALL_MODULE Sub
 DEFAULT ABORT
```

3.9.12 While Loops

The while command is a standard while loop, though using a word rather than boolean test. It has the syntax

WHILE *b* DO {R} *C*

The test expression should be a word of length 1. If the test (b) evaluates to #b1 the body (C) is executed and the test reperformed. If it evaluates to #b0, control exits from the loop. Abortion occurs if X, Z or U values are encountered. A non-executable loop invariant (R) is included for verification purposes. The command

```
WHILE run_flag DO {I} CALL_MODULE Run
```

executes the module Run until the state variable run_flag is set to #b0.

Chapter 4

The Host Program

In this chapter the host program is discussed. Initially the model used to describe the microarchitecture and the structure required of the host program to enforce this model is given. This is followed by details of sugaring to the SPOOK language which is used to produce this structure and allow a more concise description to be described. Ways that the host program can be preprocessed to reduce the amount of repeated work are then discussed. Finally, and of major importance, algorithms for collapsing the host program for particular microinstructions are given.

4.1 Introduction

The host program is a formal description of the microarchitecture. It is this description that allows the system to be universal. To verify code for a different microarchitecture does not necessitate any of the system being rewritten. Instead a new host program is provided. The microarchitecture is described as a finite state machine driven by a single universal clock. The execution of a microinstruction involves a single transition from one state to the next. The whole state of the machine is held in a collection of flip flops and registers modelled by state variables. No other state storing device may be created. The registers and flip flops are joined by combinatorial logic. Named signal wires are modelled by local variables. These are transient, retaining their value for only a single clock cycle. They cannot be used to preserve state. Input and output ports to and from the external environment are modelled by history variables.

The new state consisting of the values of the state variables, and the values output at a particular time depend only on the previous state and the values of the input signals at that time. Thus, the behaviour described by a host program is as follows:

$$\text{state}(t+1) = F(\text{state}(t), \text{inputs}(t))$$

$$\text{outputs}(t+1) = G(\text{state}(t), \text{inputs}(t+1))$$

where

- **state** gives the values of the state variables at a given time.
- **inputs** gives the values of the input variables at a given time.
- **outputs** gives the values of the output variables at a given time.
- **t** gives the time of the transition of interest.
- **F** and **G** are deterministic functions.

This model is produced by using a fixed structure to describe the host program, rather than by introducing special constructs into the SPOOK language. This keeps the semantics simple which helps in the verification process. A sugaring mechanism is used to enforce this structure. The user provides a *source host program* using the sugared syntax. This is converted by the system into the *expanded host program* which is in the pure SPOOK syntax.

4.2 The Expanded Host Program

The expanded host program should have the structure outlined below.

```
BLOCK
  $(
    Assignments to locals
    Checks that locals are well-defined
    assignments to outputs
    parallel assignment to state variables and time
  $)
```

This structure must be adhered to as it ensures that the desired model of a clocked finite state machine is obtained.

4.2.1 The Block

At the outer level of the expanded host program is a block command. It represents the single clock cycle. It assigns U values to all the locals of the machine at the start and end of the cycle. This ensures that the values assigned to locals within a block cannot propagate outside. Also, there are no assignments to ghost variables, so state cannot be preserved in that way. Furthermore, the values of inputs can be only referenced at the current time and the values of outputs not at all within the block. This ensures that the total state of the machine is stored in the state variable set.

4.2.2 Local Assignments

The first actions within the block are to make assignments to locals. These are performed in sequence so the ordering is important. In general, each local should have only a single assignment made to it, though there are certain controlled situations, described later, where this may not be so. The local assignments represent the combinatorial logic, setting up values on all the internal wires. Each state variable has associated with it a local which represents its input line. These have the same name as the state variable except that they are primed. For example, the state variable *sv* would have associated local *sv'*. The value to be stored in a state variable is first set up on its input line and transferred to the state variable only when the clock tick occurs. Similarly, each output has a primed local representing the wire inside the device connected to the port.

4.2.3 Well-Defined Checks

The locals may be assigned U values on some signals due to failures of various kinds occurring in the combinatorial logic. These are not trapped when the assignment is made for three reasons.

- It simplifies the semantics of assignment and consequently of the collapsing and verification condition generation processes.
- It makes the checks explicit, so that the user can tell from the code which locals/expressions may still hold U values after collapsing has taken place. Thus, possible ways in which the particular microinstruction may fail if not protected by its environment are identified.
- It provides a simple way in which race conditions can be trapped. This is dealt with in more detail later.

These U values must be trapped before any assignments to outputs or state variables are made, since otherwise the assignments could be illegal. The check command in conjunction with the `WD` operator is used for this purpose. A check on the well-definedness of each local is made, each having the form

    CHECK (WD *local*)

If any fail, then the check will cause the expanded host program to abort. This would prevent any verification attempt from being successful.

4.2.4 Output Assignments

Next a sequence of output assignments are made, updating the history of the outputs according to values computed by the locals. The outputs are set during the clock cycle once a value has propagated through the combinatorial logic, rather than on the clock tick. The value set during a cycle is that associated with the cycle.

The put operator is used to update the output values. A time variable `time` records the current time and is updated on each clock tick. The only history location affected is that indexed by the current time. Each assignment has the form

```
out == PUT(out; time; new_val)
```

This assigns to variable *out* its old history value, modified at the time given by variable `time` with some new value *new_val.*

4.2.5 The Clock Tick

The final action within the clock cycle is the clock tick when the machine state is changed. This is modelled by a parallel assignment to the state variables. The variable `time` is also incremented. This is the only place where this variable is altered. Each assignment assigns the value of the associated local to a state variable. These assignments reflect the action of the clock tick on flip flops. If a Z signal is input to the flip flop, its value is unchanged. This is modelled using the merge operator. The assignments have the form

```
sv := MERGE (sv; sv')
```

The use of the merge operator means that the semantics of assignment does not have to be altered for state variables in order to model this affect. Also, rewriting on the merge expression can be performed. For example, if the local is guaranteed not to hold Z or U signals, then the assignment is equivalent to

```
sv := WS sv'
```

The parallel assignment has the form

```
PAR
 $(
   sv1   := MERGE(sv1; sv'1)
                .
                .
   svn   := MERGE(svn; sv'n)
   time := INC time
 $)
```

These are actually assignments, similar to those of an imperative language such as Pascal, and are not rules as in BSPL which attach names to bundles of wires.

4.2.6 Simplicity

The structure of the expanded host program is very simple, consisting of a sequence of assignments and checks within the outer block. In particular there are no loops or branches. Despite this simplicity, complex architectures such as the Orion's can still be described within the framework. The simplicity is important as it aids the automatic collapsing of the host program as described later. It also means that all the annotations can be placed in the skeleton *before* the verification commences. This is only possible because no annotations are required within the blocks which are combined with the skeleton to give the high level program.

4.3 The Source Host Program

It is important that the host program does have the structure described in the previous section. Moreover, the description is not very concise and much of the structure could be generated automatically. For these reasons, the host program is given to the system in a much more sugared form and is automatically transformed into the above structure. The sugaring described below allows the source host program to be very concise and have a similar feel to a BSPL step function.

4.3.1 Block

The outer block command is always present and so can be added automatically.

4.3.2 Semicolons

Semicolons used as sequencing commands may be omitted from the source host program if they occur at the end of a line. As with BSPL, if the code is still to be parsed with this abbreviation, infixed expression operators may not occur as the first symbol of a line.

4.3.3 Time

The assignment to `time` in the parallel assignment is always present and so can be added automatically. The source host program should make no assignments to `time`. Allowing the system to add them automatically ensures that time flows correctly.

4.3.4 Checks

The set of locals is known, or at least can be determined statically. Thus, the well-defined checks which are necessary for all locals can be added automatically. This ensures that checks are made that no local holds a U signal, including those associated with state variables and outputs. Any U signals generated by the combinatorial logic will be trapped. For each local, l, the following command is added:

```
    CHECK (WD l)
```

4.3.5 State Variable Assignments

The state variable assignments have a predetermined format. They assign to each state variable the value of its associated local variable. The name of the local is determinable from the name of the state variable. Each of these assignments can be generated from the state variable declarations. Since the time assignment is also added automatically, none of the parallel assignment need be given in the source host program.

The name of the state variable can be used in the left hand side of the assignments to the associated local, to emphasise the fact that the assignment is ultimately made to the state variable. Primed locals, representing input wires should only appear in a left hand context in the parallel assignment, and so should not be used in the source host program. The "`:=`" form of assignment may be used as a reminder that the assignment is ultimately to a state variable. This is not rigidly enforced.

In the source form,

```
    sv := e
```

where `sv` is the name of a state variable, becomes

```
    sv' == e
```

The name *sv'* is generated by the system by adding a prime (') to the state variable name under consideration. An assignment

```
    sv := MERGE(sv ; sv')
```

is then placed in the parallel assignment.

4.3.6 Casts

The cast operators `SW` and `WS` do not need to be given. These can be determined from the context, since all state variables are declared as such.

4.3.7 Output Assignments

The output assignments can be omitted in a similar way to state variable assignments, replacing the name of the associated local in its assignment by the output name. The assignment

```
    out == e
```

becomes

```
    out' == e
```

together with the later output assignment

```
    out == PUT(out; time; out')
```

As with state variable assignment, the name *out'* is system generated from the output variable name by adding a prime character.

4.3.8 Inputs

The values of inputs are accessed using the get operator. This is similar to the state variable cast operators in that the operator can be added automatically from context. Since inputs are declared and `time` is always used to give the location accessed, the name of the input variable can be used instead of the get expression. An expression,

...*in*...

where *in* is declared as an input variable, becomes

...GET(*in*; time)...

4.3.9 Defaults

If the default case in a switch or guarded expression is just to return a word of Z signals, then it may be omitted. The expression,

```
GUARDED || g => #b_0
```

is transformed into

```
GUARDED || g => #b_0 DEFAULT Z_WORD 1
```

4.3.10 Constants

Word and signal word constants may be given using other bases. In particular, hexadecimal, octal and decimal notations may be used. With hexadecimal notation, the normal digits 0–F are used. Each octal or hexadecimal digit is converted to its equivalent binary form. The length of the word is 4 times the number of digits given for hexadecimal, and 3 times for octal representations. Octal numbers start with **#o** and hexadecimal with **#x**. The special digits X, Z and U convert to the appropriate number of X, Z or U signals. For example,

#xX $\longrightarrow$ #bXXXX

#oZ $\longrightarrow$ #bZZZ

This differs from, for example, `#x1` which represents the hexadecimal constant (`#b0001`) and not `#b1111`. The latter would be represented by `#xF`. Constants such as `#b000X` cannot be represented using the hexadecimal notation.

A decimal number is converted into a corresponding binary word having a length dependent on the context. Decimals can only be used in positions where the length is statically deducible. This means that, for example,

```
0 .= 0
```

would not be allowed, since the length rule for equality only requires that the operands be the same length. On the other hand, the expression

```
#b0 .= 0
```

would be allowed, since the decimal is forced to be of length 1 by the other operand.

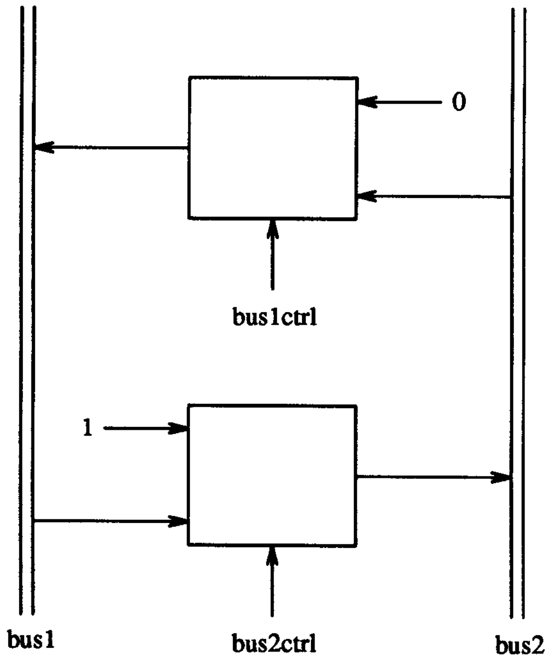

Figure 4.1: A simple feedback loop

4.3.11 Feedback

Consider the situation illustrated by the simple example of Figure 4.1. Here, two buses—`bus1` and `bus2`—have connections so that data can pass between them in both directions. Either can be loaded with a constant value depending on the values of the control signals, `bus1ctrl` and `bus2ctrl`. The loop formed between the buses is not broken by registers, so that a transfer between them can occur within one clock cycle. Therefore, if both the gates are switched to make the transfer, an unbroken loop is set up causing a potentially erroneous situation. This could correspond to a flip flop being created or an unstable signal resulting. Whatever the physical manifestation, the situation is undesirable and should be detected. It should be noted that the above condition cannot be detected statically. It is not illegal for the architecture to include potential loops of this form, only that the control signals dynamically prevent the loop being activated.

This situation does occur in real hardware. In the Orion Computer, for example, such a situation exists between the D bus and AY bus through the arithmetic and logical unit. In the Orion, an unbroken loop can only occur when the D bus function is ALU (the default). This causes the D bus to take its value from the AY bus. If the AY bus takes its value from the ALU, such as when the RAMF function is specified, and one of the ALU sources is the D bus itself, the unbroken loop is created. According to the Orion Manual, the value obtained is "undefined". Such situations should be detectable by the verification process.

In BSPL this type of structure can be described using mutually dependent signals. However, no indication of the effect of a loop becoming active is given.

The description would apparently loop infinitely. No methods are given to suggest how the verification process would catch such occurrences. One possibility would be to perform a static analysis of the step function to determine the conditions when such a loop would become active, and then include dynamic checks for those conditions. The presence of mutually recursive signals could hinder the process of collapsing host programs, since without care such loops could be collapsed away, even though they might still be active.

This problem appears to have been largely neglected in the literature. It was recognised by Barrow [Barrow 84b]. In his VERIFY system, coping with it would require the solution of simultaneous equations and possibly the introduction of new state storing variables, though this was not implemented.

The structure of the host program described so far statically prevents loop situations being described. This is because, within the host program, no loop constructs such as while commands are allowed; each variable may be the subject of only one assignment, and the assignments are strictly sequenced. There is, consequently, no possibility for locals to have mutually dependent values. To allow loops to be described and dynamically detected these restrictions are relaxed in a controlled way using the feedback command. This has the syntax

`FEEDBACK` n C

where n is a fixed integer and C a command. It is an abbreviation for n copies of C. `FEEDBACK` is not a pure SPOOK command, but an abbreviation to help enforce the appropriate structure. The value n is a fixed integer, so the feedback command can always be expanded into its pure SPOOK form. In general n will be small: typically 2, as in the Orion description. Feedback commands may not be nested.

Feedback provides a controlled way in which variable values may be mutually dependent. The situations we wish to allow are those where one variable is dependent on another at some times and vice versa at others. We do not wish each to be dependent on the other at the same time. Provided the value for n is suitably large, this will be the case. The assignments to locals which occur within a potential loop, are placed within the subcommand of the feedback command. Initially all locals will have U values as set up by the surrounding block. This ensures that locals within the loop dependent upon ones textually defined later will be given U values. When a local is given a value from a source outside the loop, as happens if the loop is not active, that value will propagate to give values to the other variables. On the later iterations of the loop, the value will propagate to the previously undefined locals. As the text of the assignments to a particular local must be identical, and the only change in values involved can be from U to non-U values, once a local has been given a non-U value it will be repeatedly assigned the same value on later iterations. In particular, changing one variable's value from U to non-U cannot change a different variable's contents from one defined value to another. If a loop occurs which is left unbroken, then no non-U values will be introduced and all locals involved will be left with U values. This will be detected dynamically by the check commands and fail any verification attempt. It is this situation which requires that it not be an error for U values to be assigned to locals. It is only an error for the U value to still be present when the clock tick occurs.

The use of feedback can be illustrated using the example of Figure 4.1. This could be described in SPOOK as follows:

```
FEEDBACK 2
 $(
   bus1 == GUARDED || bus1ctrl => bus2 DEFAULT 0
   bus2 == GUARDED || bus2ctrl => bus1 DEFAULT 1
 $)
```

This expands to

```
bus1 == GUARDED || bus1ctrl => bus2 DEFAULT #b0
bus2 == GUARDED || bus2ctrl => bus1 DEFAULT #b1
bus1 == GUARDED || bus1ctrl => bus2 DEFAULT #b0
bus2 == GUARDED || bus2ctrl => bus1 DEFAULT #b1
```

Now suppose that initially `bus1ctrl` has value `#b1` and `bus2ctrl` has value `#b0`. On the first iteration

- `bus1` gets value `#bU` from `bus2`, and
- `bus2` gets value `#b1` (the default).

On the second iteration

- `bus1` gets value `#b1` from `bus2`, and
- `bus2` is assigned the value `#b1` as before.

Thus, `bus2` is given a value from outside the loop and this value is transferred to `bus1`.

If both `bus1ctrl` and `bus2ctrl` initially had value `#b1`, on the first iteration

- `bus1` would get value `#bU` from `bus2`, and
- `bus2` would get value `#bU` from `bus1`.

This would be repeated on the second iteration. Consequently, when the loop is activated, both `bus1` and `bus2` are left with U values.

The need to enclose potential loops within a feedback command makes their presence explicit, highlighting that the microprogrammer must be wary. A disadvantage is that the host programmer must determine a suitable value of n to allow data to circulate round the loop, whatever the point the non-U value is introduced. If the value chosen was too small this would be detected in the verification process provided that the erroneous situation arose since U values would be left on some wires. The maximum value that n need be is the number of assignments in the feedback command, since it is dependent upon the number of assignments making back references. If no assignment changes the value of a variable from an undefined to a defined value then the process has converged. Otherwise,

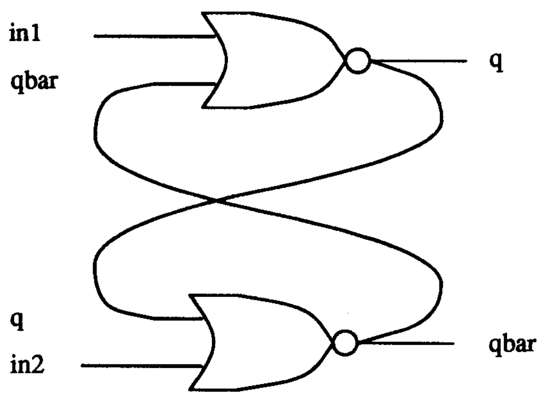

Figure 4.2: A bistable latch

one or more assignments will change a value to being defined. Since the process cannot make a variable less defined, the maximum number of steps is the number of assignments. In general the assignments can be ordered so that the value is much smaller—typically 2. An algorithm could be devised for detecting loop situations and automatically inserting the appropriate value of *n*. This has not been done.

When the feedback command is expanded the code is replicated increasing the size of the host program. This is a minor problem as the replication should not appear in collapsed host programs, leaving them just as concise. If the replication remains, then the condition used to perform the collapsing is not sufficient to prevent the loop becoming active. This could indicate an error and could easily be spotted at an early stage by examining the collapsed host program.

4.3.12 Creating State Storing Devices

A related problem is preventing state storing devices being created in the combinatorial logic. In BSPL the only legal way that state may be stored is within state variables. However, such devices could apparently be created using mutual recursion. For example, one might attempt to describe the bistable latch of Figure 4.2 as follows:

```
q    == ~(in1 | qbar)

qbar == ~(in2 | q)
```

The actual semantics of such a construct in BSPL is not clear. Complex analysis of the step function would be required to detect and prevent such situations occurring. In SPOOK such a device cannot be created, since the rules assigning values to signals are sequenced. The use of a local before its value were set would just result in the propagation of U signals, which could then be detected dynamically.

4.3.13 Local Declarations

Locals need not be declared. Any undeclared variable is assumed to be a local. The lengths of locals can be statically deduced from their context using the length rules. In general, each local will have the same length as the expression assigned to it. Therefore, by starting with the first assignment in the host program and inspecting each in turn, the length of each local can be determined. To do this the type of the expression assigned to the local is determined. The length of all variables involved in this expression should be already be known, either due to them having been declared or being locals whose length has been previously determined. If this is not so, then a local is being used before having been defined. This can occur due to feedback commands. In this case, after traversing the appropriate number of iterations, all the locals involved should have been given lengths. If some locals remain without lengths, a type error occurs. This would detect some cases when the value of n used in a feedback command was too small, though not all.

For example, consider the code

```
a == b;
b == GUARDED || g => a DEFAULT #bZZ
```

The length of b is unknown, so a cannot initially be typed. From the second assignment b will be given length 2, since that is the length of the default clause. There are no further assignments to a so it is left untyped. This occurs because the code is a loop, equivalent to enclosing it within a FEEDBACK 1 command. A second iteration would give a the type of b. The source code should be:

```
FEEDBACK 2
 $(
   a == b;
   b == GUARDED ||  g => a DEFAULT #bZZ
 $)
```

However, if the the assignments had originally been reversed

```
b == GUARDED || g => a DEFAULT #bZZ;
a == b
```

Type checking would not have detected the problem: b would be typed to length 2 and then a given that length.

A local is permitted to have a different length from the expression assigned to it when it is associated with a state variable or output. Its length will be the same as the associated variable rather than the expression . It is a type error for the length of the assigned expression to be smaller than the length of the local. If it is larger truncation may occur. This is described in Section 4.3.14.

4.3.14 Automatic Truncation and Zero Padding

BSPL allows many of the operators to automatically truncate or zero pad their operands to an appropriate length. This is not allowed in SPOOK to simplify the semantics of the operators for the collapsing process and theorem proving. It is included as a sugaring feature for the source host program. The following operators truncate the longer operand: &, |, EQV, NEQV, *. This is done using the selection operator. For example,

```
a & b
```

becomes

```
a & b[t..0]
```

where the length of a is greater than the length of b and t is one less than the length of a.

The result expressions of conditional expressions are truncated to the size of the smallest. The expression,

```
GUARDED
 || g => #b00
 DEFAULT #b001
```

would be converted to

```
GUARDED
 || g => #b00
 DEFAULT #b001[1..0]
```

The following operators coerce the arguments to be the same length by zero padding (the concatenation of a zero constant) of the shorter at the most significant end: .=, .>, .<, .>=, .<=, +, -. The expression

```
a .= b
```

becomes

```
(0, a) .= b
```

where 0 is a word of zero signals with length the difference between the lengths of a and b, and the length of b is greater than that of a.

Truncation and zero padding can only be performed when the types of the expressions involved are deducible. In the guarded expression of the example in the previous section,

```
GUARDED || g => a DEFAULT #bZZ
```

no truncation can be performed since a was a local whose type was deduced from the other guarded clauses.

Truncation of the right hand sides of local assignments is also performed automatically. This only occurs where the local is associated with a state variable or output, so that its length is known to be the same as for the declared variable. It is a type error for the length of the assigned expression to be smaller than the length of the local. The assignment,

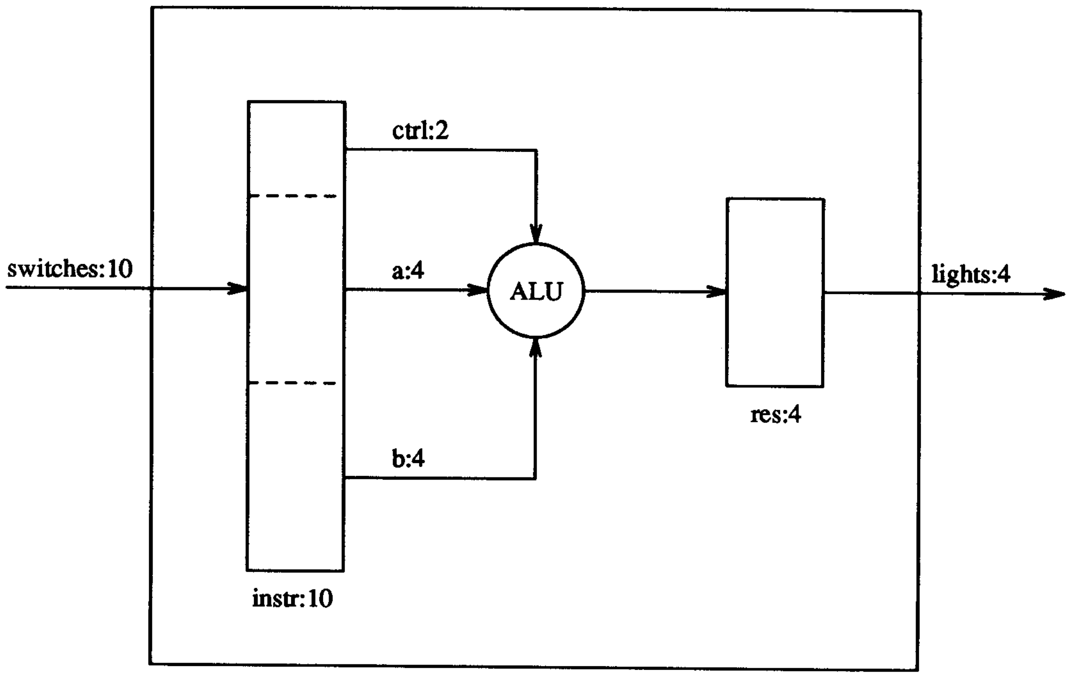

Figure 4.3: A simple device

```
a == e
```

becomes

```
a == e[t..0]
```

where t is one less than the length of a which is less than the length of e.

4.4 An Example—A Simple Device

I will use a very simple device to demonstrate the ideas presented in this chapter. This device is not intended to be a useful piece of hardware, but was designed as an example. The block diagram of the device is shown in Figure 4.3. It contains a set of ten switches, used to load an instruction register, and a set of four lights which monitor the value in an output register. The instruction register consists of three fields. The top 2 bits give an opcode which chooses between four functions. The remaining two fields of 4 bits each give operands. The four available functions are

- 0—the result is zero
- 1—add the operands
- 2—perform a bitwise or of the operands
- 3—leave the output alone.

The result is loaded into the output register at the end of each clock cycle.

4.4.1 The Source Host Program

The device has two registers—the instruction register (`instr`) and the output register (`res`). These are declared as state variables of length 10 and 4, respectively, and are the only way the device can store state.

```
STATE_VARIABLES
          instr :10
          res   : 4
```

There are 10 input lines in the form of switches, and 4 outputs in the form of lights.

```
INPUT
          switches : 10

OUTPUT
          lights   : 4
```

There are also some locals but in the source host program they do not need to be declared.

The combinatorial logic performs two tasks. It decodes the instruction register and performs the appropriate ALU function. Each decoded line from the instruction register is represented and named by a local. The function selection lines are represented by the local `ctrl`, and the operands by `a` and `b`. The decoding is done using the selection operator.

```
ctrl == instr[1..0]
a    == instr[5..2]
b    == instr[9..6]
```

The lengths of the three locals can be deduced from the number of signal positions selected. Thus, `ctrl` is of length 2, and `a` and `b` are both of length 4.

The ALU is represented by a switch expression on the local, `ctrl`. The value is assigned to the output register.

```
res := SWITCH ctrl
          || #b_00 => 0
          || #b_01 => a + b
          || #b_10 => a | b
```

The final case is omitted, since it requires that the register be left unaltered. This will happen automatically due to the default assignment of Z values. Since this assignment is made to a state variable it will actually occur at the end of the cycle.

The lights display the value of the output register. Since `res` is not updated until the end of the cycle, the value displayed is actually that set on the previous cycle.

```
lights == res
```

```
1   STATE_VARIABLES
2         instr     :  10
3         res       :  4
4
5   INPUT
6         switches  :  10
7
8   OUTPUT
9         lights    :  4
10
11  HOST_PROGRAM
12
13    ctrl   == instr[1..0]
14    a      == instr[5..2]
15    b      == instr[9..6]
16
17    res    := SWITCH ctrl
18                 || #b_00 => 0
19                 || #b_01 => a + b
20                 || #b_10 => a | b
21
22    lights == res
23    instr  := switches
```

Figure 4.4: The source host program for the simple device

At the end of each clock cycle, the instruction register is updated from the switches.

```
instr := switches
```

The complete program is given in Figure 4.4. The line numbers are not part of the program, and are given here for reference purposes only.

4.4.2 The Expanded Host Program

In the expanded host program the locals must be declared including those generated by the system and associated with state variables and outputs—`res'`, `instr'` and `lights'`. The types are deduced from the assignments. `ctrl`, `a` and `b` each have select expressions on the right hand side of the assignment, and so by the length rules, the length is the number of signals selected. `res'`, `instr'` and `lights'` each inherit the length of their associated declared variables.

```
LOCALS
        ctrl    :  2
        a       :  4
        b:      :  4
        res'    :  4
        instr'  : 10
        lights' :  4
```

The whole of the body of the program is surrounded by a block command. Each of the commands within it are separated by semicolons. Left hand side occurrences of state variables are enclosed in **SW** expressions which transform their values to kind word. Lines 13 to 15 of the source host program given in Figure 4.4 become

```
ctrl == (SW instr)[1..0];
a    == (SW instr)[5..2];
b    == (SW instr)[9..6];
```

The assignment to the state variable **res**, is transformed into an assignment to the associated local. The decimal 0, used in the switch, is expanded into its binary form. Also, the addition expression is truncated to the appropriate size, losing the carry out signal. A default case is added, assigning a word of Z values. This will ultimately cause the state variable **res** to be left unchanged if activated. Lines 17 to 20 are converted to the following assignment:

```
res' == SWITCH ctrl
          || #b_00 => #b_0000
          || #b_01 => (a + b)[3..0]
          || #b_10 => a | b
          DEFAULT Z_WORD 4;
```

The assignments to the lights and instruction register in lines 22 and 23 are also converted to local assignments. The reference to the switches is converted to a get expression at the current time, given by the variable **time**.

```
lights' == SW res;
instr'  == GET(switches; time);
```

The treatment of the lights illustrates that right hand occurrences of a state variable in the source host program refer to the value set on the previous cycle rather than the new value which is held in the associated local.

Checks that each of the locals do not hold U values are added.

```
CHECK (WD ctrl);
CHECK (WD a);
CHECK (WD b);
CHECK (WD res');
CHECK (WD instr');
CHECK (WD lights');
```

The assignment to the output `lights` (as opposed to its associated local) is then made. This updates the history value at the current time to the value of the associated local.

```
lights == PUT(lights; time; lights');
```

Finally, a parallel assignment simulates the clock tick, making the assignments to the state variables and updating time. The new value of the state variable is the merge of its old value and the value on its local. In the positions that the local has Z values, the old value of the state variable will be preserved.

```
PAR
 $(
   instr  := MERGE(instr; instr')
   res    := MERGE(res; res')
   time   := INC time
 $)
```

It is worth emphasising here that the "`:=`" symbol is used to mean assignment, whereas in the source form apparent assignments to state variables do not actually occur until the end of the cycle.

The complete expanded host program for the simple device is given in Figure 4.5.

4.5 Preprocessing the Host Program

The expanded host program is very verbose. Some preprocessing can be performed to make it more compact, while leaving it in the pure SPOOK language. Applying such preprocessing is advantageous in three ways:

- the resulting program is more readable,
- much repeated work can be avoided when producing collapsed host programs, and
- certain correctness properties can be noted from the preprocessing. In particular, if the check on the well-definedness of a local is removed by the preprocessing, then that local will always be given defined values.

Two preprocessing techniques are used:

- Check removal, and
- Expression Simplification.

Of these, the former is the most profitable technique in these circumstances.

```
STATE_VARIABLES
        instr:    10
        res:       4
INPUT
        switches: 10
OUTPUT
        lights:    4
LOCALS
        ctrl:      2
        a:         4
        b:         4
        res':      4
        instr':   10
        lights':   4
TIME
        time

HOST_PROGRAM
  BLOCK
   $( ctrl     == (SW instr)[1..0];
      a        == (SW instr)[5..2];
      b        == (SW instr)[9..6];
      res'     == SWITCH ctrl
                   || #b_00 => #b_0000
                   || #b_01 => (a + b)[3..0]
                   || #b_10 => a | b
                   DEFAULT Z_WORD 4;
      lights'  == SW res;
      instr'   == GET(switches; time);

      CHECK (WD ctrl);
      CHECK (WD a);
      CHECK (WD b);
      CHECK (WD res');
      CHECK (WD instr');
      CHECK (WD lights');

      lights   == PUT(lights; time; lights');

      PAR
       $( instr := MERGE(instr; instr')
          res   := MERGE(res; res')
          time  := INC time
       $)
  $)
```

Figure 4.5: The expanded host program for the simple device

4.5.1 Removing Checks

Each local has an associated check command for its well-definedness which ensures that an error has not occurred whilst computing its value. Often the structure of the host program ensures that a particular local cannot be given a U value. The simplest such case is if the local takes its value directly from a U-free constant. State variables are also unable to return U values, so a local assigned its value from a state variable will be U-free. The semantics of many of the operators ensure they will not return U values provided certain restrictions on their operands, such as not containing U, Z or X values, are guaranteed. With a fairly simple analysis of the host program using rules about the operators, those locals which cannot be given U values can be determined. The associated checks will always succeed and will be equivalent to skip commands. They can be removed from the host program.

In order to determine which checks can be removed, the local assignments are analysed and three sets are determined.

- the set of locals which definitely will not be assigned U signals (*Ufree*)
- the set of locals which definitely will not be assigned Z signals (*Zfree*)
- the set of locals which definitely will not be assigned X signals (*Xfree*)

The latter two are needed in the computation of the first.

The algorithm

Initially all three sets are empty. The expressions assigned to each local are examined in the order the assignments appear in the host program. If the expression could under no circumstances evaluate to a U containing value, then the associated local is added to *Ufree*, and similarly with Z values to *Zfree* and X values to *Xfree*. The currently computed values of *Xfree*, *Zfree* and *Ufree* may be used to determine these facts.

Since the assignments are sequenced and that is the order in which they are processed, an iterative algorithm is not required. Due to the structure of the host program, once a non-U value is assigned to a local, a later command will not make a different assignment. It may be that by a later assignment, more information is known—new locals may have been added to the sets—and so the local might be added when it had previously not been in a set. Locals will never have to be removed from the sets.

To determine whether an expression is guaranteed not to return X, Z or U values, rules concerning the SPOOK operators are appealed to. The full set that are currently used in the SPOOK system are given in Appendix B. They give the conditions on the operands of an operator under which it will never return a U (or Z or X) value. The conditions generally require that some of the operands are non U or Z or X, so a recursive algorithm is used. If the conditions of the appropriate rule cannot be met, or there is no such rule, it is assumed that the expression being tested might return a U (or Z or X) value. If the expression is a local, then the sets

$$\textbf{UFREE.1}\quad \frac{w \text{ contains no U signals}}{\texttt{|- FREE\_FROM\_U}\ w}$$

$$\textbf{UFREE.2}\quad \texttt{|- FREE\_FROM\_U (SW}\ e\texttt{)}$$

$$\textbf{UFREE.4}\quad \frac{\begin{array}{l}\texttt{|- FREE\_FROM\_U (}e\texttt{)}\\ \texttt{|- FREE\_FROM\_Z (}e\texttt{)}\\ \texttt{|- FREE\_FROM\_U (}f\texttt{)}\\ \texttt{|- FREE\_FROM\_Z (}f\texttt{)}\end{array}}{\texttt{|- FREE\_FROM\_U (}e\ \texttt{+}\ f\texttt{)}}$$

$$\textbf{ZFREE.1}\quad \frac{w \text{ contains no Z signals}}{\texttt{|- FREE\_FROM\_Z}\ w}$$

$$\textbf{ZFREE.2}\quad \begin{array}{l}\texttt{|- FREE\_FROM\_Z (}e\ \texttt{+}\ f\texttt{)}\\ \texttt{|- FREE\_FROM\_Z (SW}\ e\texttt{)}\end{array}$$

Figure 4.6: Parts of some of the rules for determining the well-definedness of expressions

Ufree, *Zfree* and *Xfree* are consulted. Once the analysis is completed, the checks of the well-definedness of all locals appearing in *Ufree* are removed.

Examples

Consider the assignment

```
a == #b_010 + (SW b)
```

Five rules are required to determine whether the local a could possibly be assigned U values. These are given in Figure 4.6. No analysis of X values is required.

Rule ZFREE.2 states that the addition operator will never return a Z value and so a may be added to *Zfree*. According to Rule UFREE.4 it will not return U values provided that neither operand returns U or Z values. The first operand is a constant which has no U or Z signals, so it will not return U or Z values by Rules UFREE.1 and ZFREE.1. Likewise, by Rules UFREE.2 and ZFREE.2, the second operand will not, since the cast operator, SW, can never return a U or Z value. Thus, the expression will not return a U value, and a can be added to *Ufree*.

Now consider,

```
d == #b_010 + c
```

where c is a local. Following the above reasoning, we see that whether the expression returns U values depends on c being free from U and Z values. Since it is a local, we consult *Ufree* and *Zfree*. If c is in both, then we deduce that the expression cannot return U values, and add d to *Ufree*. If it does not appear in one or the other, then we cannot be certain that the expression will not return a U value, so d is not added to the set.

4.5.2 Expression Simplification

Expressions within the expanded host program are simplified. This is done by

- constant expression evaluation, and
- the application of rewrite rules.

Constant expression evaluation consists of evaluating any subexpressions consisting only of constants, and replacing them with the resulting constant value.

Rewrite rules are pairs of functions. A test function determines whether the rule is applicable to a given expression and a replacement function returns the new expression. Lists of rewrite rules are applied together. A depth first search of the expression is performed. At each node, the subexpressions are first rewritten, and then the rewrite rules are applied in turn to the resulting expression. If any rewrite is successful, then the whole list is reapplied. Each rewrite rule is applied to the node by first applying the test function, and if it is successful replacing the node by the result of applying the replacement function to it. The full list of rewrite rules used is given in Appendix C. They are described in more detail in Section 4.6 which deals with the collapsing of host programs where expression rewriting is of greater use.

On the whole, expression simplification is liable to have only a minor effect on the host program since the host programmer is unlikely to have used complex expressions if there were simpler alternatives. The expanding process can introduce candidates for simplification, however. Moreover, if the host is to be collapsed many times with different seed values, as is so when verifying a large microprogram, any simplifications made at this stage, however small, will be worth the effort.

4.5.3 Other Techniques

Other preprocessing techniques could be used. One notable technique would be to apply the full collapsing algorithm, given in the next section, to propagate the values assigned to locals forwards. In general this would expand the size of the host program, rather than reduce it, due to large expressions being generated. The resulting code would be largely unreadable. Also, it would not necessarily save repeated work. Expressions would be propagated to multiple places, and so expression simplification would have to be performed many times on the same expression. The propagation of local values might be of value if used in a more controlled way such as if the only expressions which were collapsed forwards were constant values, or small expressions which would only be propagated to a few locations. This would require further research, however, and has not been implemented.

4.5.4 An Example

Consider once more the example of the simple device. No expression simplification can be performed on the host program due to its simplicity. Check removal can be performed. The instructions are scanned in order.

```
ctrl == (SW instr)[1..0]
a    == (SW instr)[5..2]
b    == (SW instr)[9..6]
```

Since state variables cannot return U values, and selection will only return U values if its argument does, none of `ctrl`, `a` and `b` can be assigned U values. The checks for their well-definedness can be removed. By similar reasoning, it can be seen that they will not be given Z values.

```
res' == SWITCH ctrl
          || #b_00 => #b_0000
          || #b_01 => (a + b)[3..0]
          || #b_10 =>  a | b
          DEFAULT Z_WORD 4;
```

None of the explicit alternatives in the switch expression assigning to `res'` can return U values, though the last will return Z values. The first is a non U constant. In the second and third, the addition and bitwise or operators would return U values only if their arguments, `a` and `b`, returned a U or Z signal. We have already determined that this is not so. The final case just returns Z values and so again is free from U. We must also consider the control expression of the switch. We know that `ctrl` cannot return U or Z values, though it could contain X signals. If this occurred, by the semantics of the switch operator, a U word would result. Thus, `res'` could be given both Z and U signals. Its check cannot be removed.

```
lights' == (SW res);
instr'  == GET (switches; time);
```

`lights'` takes its value directly from a state variable, and `instr'` from a get expression, neither of which can contain U values. The corresponding checks can be removed.

The resulting host program, less the declarations, which are unaffected by preprocessing, are given in Figure 4.7.

4.6 Collapsing the Host Program

The host program defines the effect of running the micromachine from any state to the state after the subsequent clock tick. Given information about the initial state or the values of inputs, specialised microinstruction specifications can be automatically produced which describe the effect of that particular microinstruction. This can be done automatically by a simple process of constant folding and rewriting. The result is a concise description of the microinstruction. For microarchitectures the most useful state information to know is that which indicates the microinstruction to be executed. This could be the value of the program counter if the microprogram is held in read-only memory. Alternatively, it could be in the form of a program counter value and a value at the indicated address. A third possibility arises with

```
HOST_PROGRAM

BLOCK
 $(
   ctrl     == (SW instr)[1..0];
   a        == (SW instr)[5..2];
   b        == (SW instr)[9..6];

   res'     == SWITCH ctrl
                 || #b_00 => #b_0000
                 || #b_01 => (a + b)[3..0]
                 || #b_10 => a | b
                 DEFAULT Z_WORD 4;

   lights' == SW res;
   instr'  == GET(switches; time);

   CHECK (WD res');

   lights   == PUT(lights; time; lights');

   PAR
    $(
      instr   := MERGE(instr; instr')
      res     := MERGE(res; res')
      time    := INC time
    $)
 $)
```

Figure 4.7: The preprocessed expanded host program for the simple device

```
BLOCK
 $(
   lights    ==  PUT(lights; time; SW res)
   PAR
    $(
      instr  := MERGE(instr; GET(switches; time))
      res    := #s_0111
      time   := INC time
    $)
 $)
```

Figure 4.8: A collapsed host program for the simple device

pipelined architectures such as the Orion, where the value is that in an instruction register.

Other information known to hold before the microinstruction of interest is executed can also be used to specialise the host program further. This is especially useful when the architecture uses (pipeline) registers to store control information. For example, in the Orion the memory operation executed is determined from the value held in a pipeline register so this part of the state can be usefully used to collapse the host program. The values of local variables cannot be used to seed the collapse of the host program, since their values are internal, and are in general derived from state and input values.

As an example of a collapsed host program, consider the preprocessed host program of the simple device of Figure 4.7 when executing a bitwise *or* (opcode `#b10`) of `#b0101` and `#b0011`. To perform this operation the instruction register will hold the value `#s_0101_0011_10`. It is this value which is used to perform the collapsing. The resulting collapsed host program is given in Figure 4.8. It is significantly more concise than the preprocessed host program giving a precise description of the action of the device when the instruction register has the given value. It can be clearly seen that it

- loads a new value into the instruction register from the switches,
- displays the previous value held in the output register on the lights, and
- computes the result of the *or* operation (`#s0111`) placing it in the result register.

This example will be examined in detail later in this section.

Collapsed host programs are useful in several ways.

- They give a means for microprogrammers to check that their understanding of a microinstruction is correct, ensuring that no unforeseen side effects occur. This can be done at an early stage in the development cycle, allowing mistakes to be discovered quickly.

- They may be used as an aid to understanding a microprogram by someone other than the original microprogrammer.
- They can be combined with a skeleton to form a register transfer version of the microprogram as a whole.
- They reduce the complexity of the verification task at an early stage, making the problem of verifying code on complex hosts more tractable.

4.6.1 The Algorithm

The host program will have the form

```
BLOCK $( $C_1$; $C_2$;... $C_n$ $)
```

where each C_i will be one of: an assignment to a local; an assignment to an output; a check command, or a parallel assignment command. Since only these simple commands are present the collapsing process is straightforward. Each of the commands is examined in turn, substituting in any known values for variables and simplifying it as much as possible. If it is a local assignment, then if possible a new substitution is noted, and the assignment removed. The details of the process are given in Algorithm 4.6.1. It is assumed that the information known about variables is held in an association list (a-list) datastructure: a list of pairs consisting of the variable name and its value. Sets *Xfree*, *Zfree* and *Ufree* are maintained, as in the preprocessing stage, to record which locals will definitely not contain X, Z and U signals. These sets are initially empty.

Algorithm 4.6.1 (Forming Collapsed Host Programs)

```
For $i$ = 1 to number_of_commands do
    1. Replace each occurrence of any variable within the expressions of
       command $C_i$, for which a substitution exists in the a-list, with the
       value indicated by the most recently added such substitution.
    2. Simplify the expressions within the command $C_i$ using the informa-
       tion recorded in the sets Xfree, Zfree and Ufree.
    3. Simplify the command $C_i$ using command rewrite rules.
    4. Take the appropriate action from below, depending on the form of
       the resulting simplified command $C_i$.
        • SKIP,
            – Remove it from the host program (unless it is the only
              command).
```

- *V* == *E*

  where V is a local variable and E is not a GUARDED *or* SWITCH *expression*

  - *Add a new substitution (V, E) to the front of the a-list. This means that later occurrences of V in the host program will be replaced by E until a new assignment is encountered.*
  - *Delete the assignment from the host program.*

- *V* == *E*

  where V is a local variable and E is a GUARDED *or* SWITCH *expression*

  - *Remove the entries for V (if any exist) from the a-list*
  - *Examine E to determine whether V is guaranteed not to return X, Z or U values and update the sets Xfree, Zfree and Ufree accordingly.*

The collapsed program resulting from this algorithm will have an identical effect on the state variables and outputs of the microarchitecture from a given state, as would the host program, provided the initial state satisfies the information used to perform the collapsing.

Conditional Expressions

Assignments of guarded and switch expressions are considered too complex to be collapsed forwards. Since there may be earlier substitutions to the variable in the a-list, the a-list cannot be left unchanged. If this were done, the earlier substitutions would continue to be used, even though the value was no longer valid. Thus, in this case any previous substitution for the variable is removed. Since the only situation where there will be more than one assignment to a variable is within feedback loops, there will normally not be a previous substitution.

All commands are simplified, including those assigning conditionals. Even if a command originally assigns a conditional expression, this conditional is likely to simplify to a form which can be collapsed forwards. Using the heuristic that only conditional expressions are too complex to collapse forwards appears to work well in practice, given a suitable set of expression rewrite rules. On the whole, the resulting collapsed host programs are readable. Furthermore, if all assignments were collapsed forwards the resulting program would, in general, be very difficult to understand.

The algorithm only avoids collapsing local assignments forwards if the conditional expression is assigned at the outer level and not if they occur as a subexpression of the assigned expression. For example, an assignment

```
a == b, GUARDED...
```

would be collapsed forwards. This situation need not arise, however. The SPOOK language was designed with the intention that conditionals would only be used at the outer level, mirroring the guarded rules of BSPL. This could easily be enforced syntactically. When describing microarchitectures this is a very natural style of programming. Furthermore, any host program containing inner conditionals could be transformed to one without, by creating new locals. The above assignment could, for example, be replaced by

```
new == GUARDED...
a   == b, new
```

This corresponds to naming the wires out of a functional unit. Nested conditionals could be allowed, and do occur in the Orion host program, since they cannot cause a problem.

The expanding process can introduce inner conditionals. If a conditional expression has a length larger than the variable that it is assigned to, then it will be truncated using a selection expression. Expressions of the form

(SWITCH...) [*n*..*m*]

are introduced. However, rewrite rules are provided which move selection operators inside conditionals. Consequently, the preprocessing process will transform such expressions from the expanded host program before collapsing is performed. Since assignments of conditionals are not collapsed forwards, the collapsing process itself cannot introduce inner conditionals so they should not occur.

Substituting Values for Variables Within Commands

In Step 1 of the collapsing algorithm variables are replaced by their values within the expressions of commands. Each expression within the command is examined in turn, and the substitutions made within that expression. This is performed using a depth first search of the expression replacing any variable encountered by its value from the a-list if a substitution for it exists. This process has the effect of making a parallel substitution of the most recent values known for each variable within the expression. These values could be symbolic values.

Expression Simplification

Expression simplification when collapsing the host program is performed in a similar way to that used during the preprocessing stage, incorporating both constant expression evaluation and rewriting.

The rewrite rules used are fairly ad hoc and are by no means complete. They are, however, sufficient to collapse to a readable extent the examples considered, and in particular the Orion host program. The rules required to collapse these examples are likely to be equally useful for other host programs, since the main constructs used are liable to be similar. This section describes some of the rewrite rules used to give their flavour. In particular rules for collapsing selection, subscription and conditional expressions will generally be required and will be responsible for the bulk of the collapsing.The full set of rules used to collapse the Orion Computer host program is given in Appendix C.

The selection operator is frequently used in host programs and several rules (EXP.21 – EXP.29) simplify select expressions. For example, by Rule EXP.25(Select-concat-low), expressions of the form

```
(e1, e2)[s1..s2]
```

can be simplified to

```
e2[s1..s2]
```

if all the selected signal positions fall within the length of e_2.

The conditional expressions are also used frequently and are good candidates for collapsing. For example, in a guarded expression, any clause with a `#b0` guard can be removed using Rule EXP.38(Guarded-zero). In a switch expression, if the subexpression switched upon is a constant, then it can be simplified to the appropriate result expression using rule EXP.33(Switch-constant). Due to the nature of the conditionals, not all the possible result expressions need to be evaluated to determine the actual result so conditionals can often be simplified.

Constructs which are commonly used are read only memories and lookup tables. These are modelled using the subscription operator. The read only memory or table contents are represented by a constant. Thus, expressions of the following form are frequent.

```
#b...!n a
```

Since these structures are often used to decode instructions, it is very important that they can be simplified during the collapsing process. The simplest case that occurs is when the address is also a constant. This situation arises in Gordon's computer, where the control ROM is modelled by a constant (stored in the local `rom`) subscripted by the microprogram counter. The latter is used to seed the collapsing process, and so is converted to a constant during the collapsing process. The expression

```
rom !29 (SW mpc)
```

represents the microword to be evaluated. In this case both `rom` and `mpc` have known values. Therefore, the above expression is converted to a constant expression. Consequently constant evaluation yields the constant representing the microword. A similar situation arises in the Orion host program to decode both the `ca_ir_sfunc` control word field and the destination of the ALU result.

A more complex situation arises when the address consists of a concatenation of expressions, only some of which are constants. The decoding of the memory operation is performed in this way in the Orion computer. The expression which arises has the form

```
#b100_100...000_000 !3 (mfunc, SW raw_wflt, SW raw_rflt)
```

mfunc takes its value directly from one of the control word fields so a constant value becomes known for it during the collapsing process. raw_wflt and raw_rflt are state variables so their values will not necessarily be known, and often do not affect the result of the expression due to repetition within the constant. The subscription becomes,

```
(#b_100...) !3 (#b..., SW raw_wflt, SW raw_rflt)
```

If some bits of the address are fixed, then certain addresses cannot possibly be selected. The expression can be simplified by removing the words at the impossible addresses, and removing the constant from the address. This is performed by Rule EXP.18(Subscript-concat1)

Consider when the required memory operation for the Orion is IDLE so that mfunc has value #b000. The address becomes

```
#b000, SW raw_wflt, SW raw_rflt
```

Only the first four addresses could possibly be chosen (raw_rflt and raw_wflt both have length 1), so the expression is simplified to

```
(#b_000_000_000_000) !3 (SW raw_wflt, SW raw_rflt)
```

In this case, the expression can be simplified even further since all the alternatives are the same. Provided the address cannot contain U or Z values, that value will be the result. This is the case in the above, since the address is formed from the concatenation of state variable values. The expression ultimately simplifies to #b000. The memory operation is decoded even though the full address is not known. This allows the host program to be satisfactorily collapsed without having to give the irrelevant values of raw_rflt and raw_wflt. A similar situation occurs at other points in the Orion host program such as decoding the operation performed by the microprogram controller. Note the importance here of the semantics of the subscription operator with respect to X signals in the address. The most accurate result possible is returned using the SHAKE operator so #b000 would still be returned, even if raw_rflt or raw_wflt did contain X signals.

Rewriting Commands

Commands are rewritten by applying rewrite rules in the same way that expressions are rewritten. This is straightforward for commands, since only simple commands with no subcommands are present in the host program. Due to the very simple structure of host programs, very few command rewrite rules are used. They are given in Appendix C. There are basically four kinds of command which occur within the block of a host program—assignments, sequencing, checks and parallel assignments. Assignments and sequences are simplified as an integral part of the collapsing process and so individual rewrites are not used. Check commands are rewritten to skip commands if the expression checked is #b1. This is performed by Rule COM.1(Check-one). They will subsequently be removed from sequences. Individual assignments within a par command can be removed by the Rule COM.2(Par-eq) if they reduce to the form,

```
a := a
```

This form will result whenever a state variable is to be left unchanged. Its input local will be assigned Z values in this case resulting in an expression

```
a := MERGE(a; Z_WORD n)
```

since `MERGE(a; Z_WORD n)` simplifies to a. This simplification would be performed during expression rewriting.

4.6.2 An Example

To illustrate the collapsing process, consider the simple device presented earlier in this chapter in a state when the instruction register holds the value #s_0101_0011_10. This will have been read in from the switches on the previous clock cycle. Each command within the block of the preprocessed host program is examined in turn, building up an association list of known information. Initially this contains a single pair giving the information about the value of the instruction register.

[(instr, #s_0101_0011_10)]

This value is substituted into the expression in the first command

```
ctrl == (SW instr)[1..0]
```

and constant folded to give

```
ctrl == #b_10
```

This is a local assignment, the left hand side of which is not a guarded or switch expression, so the assignment is deleted and a new substitution is stored.

[(ctrl, #b_10), (instr, #s_0101_0011_10)]

We now move to the next command and repeat the process. In this way, substitutions for all the locals are determined, and their assignments removed. In the assignment to res', the switch expression rewrites to a constant since ctrl, a and b are all known. It too is removed.

All but one of the check commands were removed during the preprocessing stage. The remaining command is the check for the well-definedness of local res'. A constant value is known for this and so the resulting expression,

```
WD #b0111
```

is evaluated leaving the command,

```
CHECK #b1
```

This is rewritten to a skip command and subsequently removed.

The next command is the assignment to lights.

```
    lights == PUT(lights; time; lights')
```

The value for `lights'` is substituted in, but since the command is not a local assignment, a new substitution is not made and the assignment is not deleted.

```
    lights == PUT(lights; time; SW res)
```

The parallel assignment is similarly rewritten. The resulting collapsed host program is shown in Figure 4.8. It gives a concise description of the effect of the machine when the instruction register has the value `#s_0101_0011_10`.

4.6.3 A Comparison with Other Systems

A similar collapsing process to that described here has been used in other verification systems. In all but the MIDDLE system, the approach has not been an integral part of the system. Rather it has been adopted by the verifier as a natural way to structure the problem. The individual specifications have been used to aid the verification and not the original design of the code. For example, in Hunt's proof of the FM8501 [Hunt 85] and its subsequent reproof using SDVS [Crocker 88], collapsed specifications took the form of lemmas about the effects of individual microinstructions. In both cases, similar lemmas describing the actions of instruction sequences were also proved. The proofs were conducted independently and as Crocker et al. point out "The commonality of approach is evidence that this is the natural structure of the problem" [Crocker 88]. Similar lemmas were also proved in the HOL proof of the VIPER microprocessor [Cohn 87]. In the MIDDLE system [Budkowski 78], a register transfer level program is produced from the bitstream version. This is done by effectively producing individual specifications for each word in the microinstruction store. The process is, thus, an integral part of the system. It is more limited than the other systems, including SPOOK, however, in that the only information available for the collapsing is the value of the microword. In the other systems, any state information could be used.

Chapter 5

The Skeleton

This chapter, in which the skeleton is described, forms the crux of the methods described in this dissertation. Initially, the motivation for using a skeleton is given. The facilities provided by the SPOOK language for writing them are then discussed followed by ways in which collapsed host programs can be combined with the skeleton to produce a high level version of the microprogram. Finally, ways in which this high level microprogram can be simplified are detailed.

5.1 Introduction

The skeleton is a formal version of a state transition graph of the microprogram. It is provided by the user as additional documentation of the code. It differs from a traditional state transition diagram since it is given using the high level constructs of the SPOOK language. It does not have to mirror exactly the structure of the code. This would in general be impossible since SPOOK does not have a jump command. These are prolific in microprograms. Instead, the code is tidied up to give a structured and modular version. This is a great advantage since

- it is more human readable,
- it provides the structure for the program which is ultimately verified, and
- it provides a framework upon which other documentation can be placed.

5.2 State Transition Documentation

The advantages of providing state transition documentation in general, and a skeleton in particular, will be illustrated using an example.

5.2.1 Assembly Code

Below is the part of the assembly code from the microprogram of Gordon's computer which implements the macroinstructions. Only the final column is of importance to the control flow. A `jmp n` microopereration performs an unconditional jump to location *n*. A `jze n m` microoperation causes a jump to location *n* if the accumulator is zero and to location *m* otherwise. A `jop n` microoperation jumps to the location given by adding *n* to the top three bits of the instruction register.

```
9                      jop 10
10                     jmp 0
11  rir   wpc          jmp 5
12                     jze 17 11
13  racc  warg         jmp 19
14  racc  warg         jmp 22
15  rir   wmar         jmp 24
16  rir   wmar         jmp 25
17  rpc   wbuf   inc   jmp 18
18  rbuf  wpc          jmp 5
19  rir   wmar         jmp 20
20  rmem  wbuf   add   jmp 21
21  rbuf  wacc         jmp 17
22  rir   wmar         jmp 23
23  rmem  wbuf   sub   jmp 21
24  rmem  wacc         jmp 17
25  racc  wmem         jmp 17
```

The flow of control through this structure is not obvious. It is also difficult to identify which code implements a particular macroinstruction. The instructions which implement the add instruction, for example, are at locations 13, 19, 20, 21, 17 and 18. It is not contiguous and so is difficult to follow. This situation is common with low level code.

5.2.2 State Transition Diagrams

The state transition diagram for this code, given in Figure 5.1, increases its understandability.

Each node in this diagram represents a state from which a microinstruction is executed. The associated number gives the location of the microinstruction in question. The branches give the possible transitions to new states—the executable jumps in the code. The separate paths through the code can easily be identified, including the locations executed. However, it is just an informal description. It plays no part in the verification process, and is not checked. Undetected documentation errors could occur. Indeed, the transition diagram given by Joyce for the Tamarack—a version of Gordon's computer—missed off the path implementing the SKIP instruction [Joyce 88b]. Furthermore, the diagram is unstructured and not

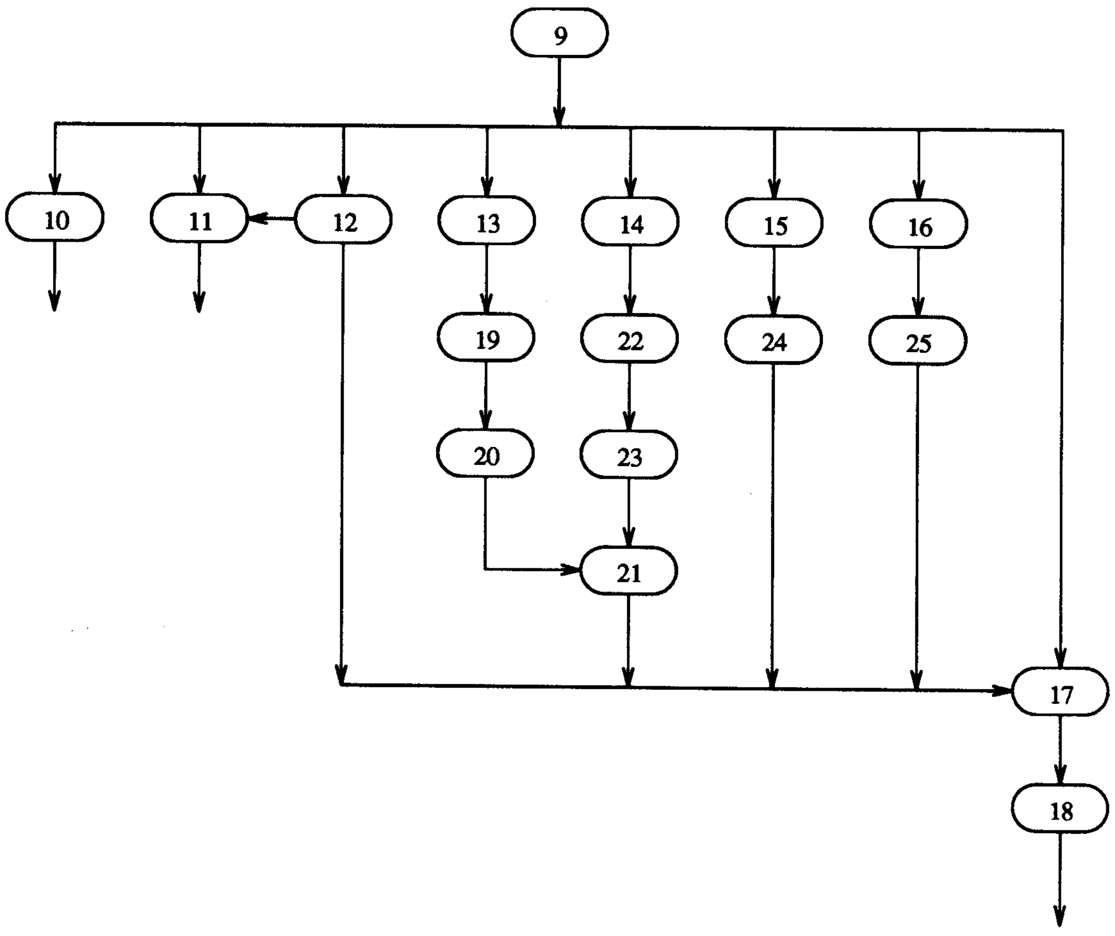

Figure 5.1: Part of the state transition diagram for the microprogram of Gordon's computer

modular. There is no indication of which code implements which macroinstruction, or what the expected action of each particular microinstruction is.

5.2.3 Skeletons

SPOOK provides language constructs for describing a state transition diagram formally as a skeleton. Part of the skeleton for the code of Gordon's computer is given below. It gives the branch which chooses the code to be executed for a particular macroinstruction.

```
SKELETON
           .
           .
    MI (SW mpc .= 9);{}
    CASE Op
     || 0 => CALL_MODULE Halt
     || 1 => CALL_MODULE Jmp
     || 2 => CALL_MODULE Jze
     || 3 => CALL_MODULE Add
     || 4 => CALL_MODULE Sub
     || 5 => CALL_MODULE Lda
     || 6 => CALL_MODULE Sta
     DEFAULT CALL_MODULE Skp
           .
           .
```

The MI command indicates the point when a microinstruction is executed and provides a condition specifying the actual state. Here, the value of the microprogram counter is expected to be 9. The microinstruction at that location is expected to be executed. The condition can be more general in the state transition diagram, since conditions on any part of the state can be given. The multiple branch of microinstruction 9 is modelled using a case command. This has a default case so if any path has been missed there will be a mismatch between the code actually executed and that given by the skeleton. This would be caught by a verification attempt. `Op` is a ghost variable having the value of the current opcode being evaluated. An annotation must be placed before the case command. In the above an empty assertion (`{}`) was used. Before verification can be attempted, such empty annotations must be replaced by non-empty ones.

Each macroinstruction is described by a separate named module and so is easily identifiable. For example, the Add module is given below. The code, which in the original microprogram involved several jumps, is presented as contiguous code in the module. Also, shared code has been split to give separate copies in each module so the path being executed has been notionally used to specify the states. Furthermore, informal comments may be interspersed in the skeleton. In the example, each MI command is commented with the assembly instruction it represents. Thus, an indication of what each instruction does is given. This means the code can be understood just from the skeleton, without having to refer to the assembler source.

```
MODULE Add
{}
 $(
   MI((SW mpc).=13);                  // racc warg jmp 19
   MI((SW mpc).=19);                  // rir wmar jmp 20
   MI((SW mpc).=20);                  // rmem wbuf add jmp 21
   MI((SW mpc).=21);                  // rbuf wacc jmp 17
   MI((SW mpc).=17);                  // rpc wbuf inc jmp 18
   MI((SW mpc).=18)                   // rbuf wpc jmp 5
 $)
{}
```

5.3 Facilities of SPOOK

5.3.1 Structure

The main purpose of the skeleton is to provide structure. The SPOOK language contains constructs specifically for this purpose. Structured branches are provided by the conditional commands and loops by the while command. These must mirror the control flow of the microprogram as modelled in the host program by assignments to the microprogram counter. The conditional commands, thus, have similar semantics to the conditional expressions, using word expressions within tests. Since the structure given by the skeleton ultimately becomes the structure of the program verified, only constructs which are amenable to verification are provided. In particular there is no jump command. The simple module facility of SPOOK also allows the microprogram to be divided into independently validatable portions.

5.3.2 Nodes

Since the skeleton describes a state transition diagram, it must use a construct to indicate the positions of nodes within the diagram. The MI command is provided for this purpose. The states represented by the node are those in which the associated condition has value `#b1`, therefore the possible states are clearly defined. Allowing an arbitrary condition to be used to specify the state is also very flexible. It does not have to, for example, just indicate the value of the microprogram counter, but can specify the values of other variables, and even include more general conditions. Furthermore, the same information does not have to be given for all nodes.

5.3.3 Decimal Conversion and Other Bases

To make the skeleton more readable, expressions may include decimal numbers in a similar way to the source host program. These represent words of the appropriate length, so can only be used in positions where the length is deducible. With the case and if commands, the guards and labels have a known length and so this information can be used to perform decimal conversion. For example, in the skeleton for Gordon's computer, the following code fragment occurs

```
CASE Idle
 || 1 =>  ⟨ idle code ⟩
 DEFAULT  ⟨ run code ⟩
```

The label 1 will be converted to #b1, since it must have the same length as the subject of the command, Idle, which has length 1.

Octal and hexadecimal forms can also be used as an abbreviation for word constants as in the host program.

5.3.4 Ghost Assignments

Since the skeleton is intended only to give the control structure of the microprogram, it is only allowed to change the state of the underlying machine implicitly using MI commands. For this reason, there can be no assignments to local, input, output or state variables within the skeleton or any of its modules. The only permitted assignments are those to ghost variables.

Ghosts are useful as more readable aliases for the value of some resource at some particular time such as the start of a cycle, and for storing the values of resources so that an annotation can compare the current value with the original. In these situations assignments can often be avoided. The value of the ghost can instead be set in the precondition. There are occasions when ghost assignments are useful. They can be used to help manipulate the skeleton into a more structured and readable form. For example, in Gordon's computer, the ghost, Idle, is used to indicate which of two modes—idling or executing—the computer is in. A ghost assignment is made at points where the mode is changed. At the microlevel, this is indicated by the value of the microprogram counter and the state of the idle light. The use of the ghost allows the skeleton to have a clean structure.

```
CASE Idle
 || 1 =>  ⟨ idle code ⟩
 DEFAULT  ⟨ run code ⟩
```

Without it the switch would have to be made on the value of the microprogram counter, to allow the code to be structured in this way. The ghost variable makes the skeleton more understandable. Ghost assignments could similarly be used to escape from while loops allowing complex control structures to be modelled.

Ghost assignments are also useful for recording traces of the values intended to be output. This was the purpose for which ghost assignment was originally suggested by Clint [Clint 73]. This would require ghost history variables which are not currently supported in SPOOK so it cannot currently be done.

5.3.5 Assertions

Assertions are placed in the microprogram at various points, to give the target specification and to aid the production of verification conditions. The skeleton gives a suitable framework from which to hang these annotations. Due to the structured nature of the skeleton, the points where assertions are required is clear and can be enforced syntactically. The assertions can also be placed during the documentation process, prior to the proof of correctness. This contrasts with the systems where the control structure is determined symbolically as the proof progresses. In such systems, the places where assertions are required is rediscovered as part of the verification process. In these systems there is no syntax directed way of placing assertions. The skeleton contains all the formal documentation of the microprogram: the state diagram and annotations which document how the microprogram works and also the target specification.

The Target Specification

The target specification describes the macroarchitecture. It is this specification that the microprogram is proved to be consistent with. It is given in the form of a partial correctness specification for the microprogram. A precondition describes conditions which must hold before execution of the microprogram. Given this precondition, a postcondition describes the intended final state.

The partial correctness statement is incorporated into the skeleton using the conventional notation of placing the conditions in curly brackets, before and after the program. The skeleton has the form

```
SKELETON
  {P}
  C
  {Q}
```

where P gives the precondition, C is the body of the skeleton and Q gives the postcondition.

The specification is partial because it only states that if the initial state satisfies the precondition, and the program terminates, then it will finish in a state satisfying the postcondition. The specification does not state that the program will terminate. This can be specified and proved separately. As with the proof of partial correctness, the techniques described so far would be relevant to this task. It is beyond the scope of this dissertation to consider questions of termination, however.

It is perhaps more apt to think of the target specification as being documentation of what the microprogram actually does, rather than as a specification of what it is intended to do. This is because the verification process proves the microprogram is consistent with the assertions, rather than proving it "correct".

Inductive Assertions

An inductive assertion [Floyd 67] must break each loop within the skeleton. In SPOOK, the only way to construct a loop is using the while command. An annotation must be placed after the keyword DO in all while loops. For example, the while loop

```
WHILE b DO {I} C
```

would require an inductive assertion, *I*, as shown.

Sequences

In theory, the only annotations required are inductive assertions. However, in practice it is convenient for assertions to also be placed between some sequenced commands. In particular, in SPOOK, they must be placed before if, case, while and module calling commands which occur as the second command in a sequence. For example, a command

```
C'; {R} WHILE b DO {I} C
```

would require the assertion R as shown in addition to the invariant I.

Module Specifications

Each module contains an individual precondition and postcondition. For example, a module,

```
MODULE wombat
  {P}
  C'; {R} WHILE b DO {I} C
  {Q}
```

requires a precondition P, and postcondition Q. This allows the module to be verified independently. This is again an advantage provided by the skeleton, which allows separate modules to be identified and individually specified.

Specification errors account for a large proportion of errors in code. Providing a formal specification alleviates this to some extent, as it removes the possibility of errors due to ambiguity. Errors in formal specifications can still occur. As with programs, it is easier to write specifications and subsequently check them if they consist of small, intellectually manageable and independent portions. This can be done naturally when implementing a macroarchitecture, since each macroinstruction can be specified separately. Since the skeleton for each macroinstruction is placed in a separate module, each can be given an independent partial correctness statement.

5.4 Expanding the Skeleton

The skeleton is used in the SPOOK system as a basis around which to produce a register transfer level version of the microprogram. Producing such a detailed though concise description of the microprogram is advantageous for several reasons.

- It allows the microprogram to be treated as a program, rather than just as data to an interpreter, during the verification process.
- It allows syntax directed verification techniques to be used.
- It provides the microprogrammer with an opportunity to visually check the microprogram at an early stage in the design cycle.
- It may be used in conjunction with a simulator to test the microprogram.
- It provides further formal documentation of the microprogram which may be used to help others, including independent verification teams, understand it.

5.4.1 Replacing MI Commands

The semantic rule for an MI instruction given in appendix A is

$$\frac{\vdash \{P\}\ \texttt{CHECK}\ e\ ;\ HP\ \{Q\}}{\vdash \{P\}\ \texttt{MI}\ e\ \{Q\}}$$

where *HP* is the host program.

This states that executing an MI command involves executing a check followed by the host program. Since the skeleton cannot directly affect state variables, outputs or locals, the effect of executing the skeleton on the microarchitecture resources is the same as when executing the host program some number of times, provided the skeleton does not abort. The abortion could be caused by an actual abort command being executed, or by the condition in one of the MI commands not evaluating to `#b1`. In either situation the skeleton programmers view of the microprogram is incorrect so abortion is the desired effect. Execution of the microprogram involves repeatedly executing the host program, which is just an interpreter for the microprogram. Therefore, if the skeleton does not abort, the behaviour displayed with respect to the microarchitecture resources will be identical to the behaviour of the microprogram. This means that validation can be safely performed on the skeleton rather than on the microprogram. Furthermore, if a formal verification attempt is successfully applied to the skeleton, then the correctness results gained will also apply to the microprogram.

Performing validation on the skeleton might at first seem to be disadvantageous since it involves extra overhead. The control flow is in effect calculated twice; once for the high level constructs and once in the resources of the microarchitecture such as a microprogram counter. However, advantages are gained, which outweigh this disadvantage.

- The documentation given by the skeleton is checked during validation.
- Individual modules of the skeleton can be validated separately.
- Annotations, such as inductive invariants, can be placed in the skeleton itself. A separate, and less understandable, mapping between annotations and states does not have to be given. Also, loops and therefore the positions where annotations are needed is made explicit.
- It is easier to discover the location of errors if working with a high level control structure.
- Collapsed host programs can be used instead of full copies of the host program,
  - radically improving the efficiency over repeatedly executing whole copies of the host program,
  - giving a decompiled, though high level version of the microprogram which can be checked by eye, and
  - removing much of the complexity caused by the large amounts of detail at an early stage in the verification process.

Since the semantics are identical, we could actually replace the MI commands by the appropriate check and copy of the host program. The resulting voluminous program would have exactly the same behaviour as the skeleton itself. Now, since each copy of the host is executed after a check command, the condition given in the check can be used to collapse the corresponding host program. The resulting program will still have the same behaviour, since the collapsed host programs have the same behaviour as the full host program, provided they are executed in a state satisfying the condition used to perform the collapsing. The state will always satisfy that condition since the check would cause abortion if it did not. Each command

MI e

can be replaced by

CHECK e ;
$\langle$ *host program collapsed with condition* e $\rangle$

5.4.2 Extracting Information from the MI condition

The only information from the condition of the MI command which is used to perform the collapsing in the SPOOK system is equality information. The condition is assumed to have the fixed form:

((SW v_1) .= e_1) & ...& ((SW v_n) .= e_n) & B

where

- the v_i are state variable names,

- the e_i are arbitrary expressions, and
- B is an optional expression of some different form.

To collapse a host program we need to know actual equivalences, but the above condition does not directly give equivalences. The `.=` operator is not an equivalence operator as it may return X or U signals. In its given form, all we know is that the condition evaluates to `#b1`. However, if that is so, by the semantics of the `.=`, `&`, `SW` and `WS` operators, each variable v_i will be equivalent to the corresponding expression (`WS` e_i). From the above expression, the set of substitutions

$\{ [v_1 \ / \ \texttt{WS}\ e_1] \ldots [v_n \ / \ \texttt{WS}\ e_n] \}$

is formed and used to seed the collapsing process.

Other information could be extracted from the MI condition, and used to perform the collapsing. For example, more general clauses such as

```
acc .< 0
```

could be replaced by `#b1`, where they occurred in the host program, if given in the condition. Also, equality information about selections, such as,

```
instr[3..0] = #b0000
```

could be used. The algorithms for doing this have not been implemented.

5.4.3 Collapsing from a Word Expression

The information used to perform the collapsing could be presented to the system in several ways, rather than as a word expression.

The simplest method would be to make the microprogram counter distinguished, and pass a single constant giving its value. This is, in essence, the technique used in the MIDDLE system. It has several disadvantages, the most obvious being that only one value can be passed. Often other parts of the state are relevant, especially for commercial architectures. For example, in the Orion computer, memory operations are requested a cycle early, so the value is stored in a state variable and is therefore relevant to the collapsing process. Furthermore, using this method the microprogram counter must be distinguished in some way. Also, the microprogram counter is not always the most relevant part of the state. Again in the Orion, the instruction to be executed is stored in a pipeline register, and so it is the value in that register that is required to collapse the host program.

A second approach would be to give a list of pairs consisting of a variable and its value. This would be similar to the form that the SPOOK system currently converts the condition to, to collapse the host program. It is not executable, however, and would not be checkable in such a natural way as a word expression. Also, it could not be easily extended to cope with more general information.

A further alternative would be to give the information as a predicate calculus assertion, extracting the useful facts in a similar way to that actually used in

SPOOK. Assertions are not in general executable, however, preventing simulation from being used as a method for validating the collapsing information.

The method of presenting the collapsing information as word expressions was preferred since

- it is flexible,
- the condition is executable and so can be validated with the rest of the program,
- the semantics of the language are kept simple, and
- the condition is in a human readable form.

5.4.4 Reusing Collapsed Host Programs

Often identical microinstructions are used in several different places throughout a microprogram. We would like to make use of this fact to reduce the amount of work done. This can easily be accommodated using collapsed host programs. The situation may arise due to the microinstruction at a particular address being used at more than one point in the skeleton, such as when code is shared between different modules. For example, in Gordon's computer, many of the macroinstructions increment the program counter. This is done by shared code held in memory at addresses 17 and 18.

```
17  rpc   wbuf  inc  jmp 18
18  rbuf  wpc        jmp 5
```

To make the modules self contained, and to reduce the complexity of the control structure, each module in the skeleton has a separate copy of these instructions.

Alternatively identical microinstructions may occur due to the same code occurring at different points in memory. In the standard Orion microcode, the code implementing each macroinstruction must reload the instruction register. For many of the arithmetic and bitwise operators this is performed by the same, though not shared, microinstruction given below.

```
CONT DZ SHR1 OR RAMA A=ir0 B=ir0 LDIR DECCA
```

If large fragments of code are used in this way, they could be split into a separate module and independently verified. For a single microinstruction, this would not be worthwhile. Instead, much repeated work can be avoided by saving the collapsed microinstructions in a table keyed on the condition used to perform the collapsing. This would mean that each collapsed host program would only be formed once. It would be a great advantage if the collapsing process was used as a firmware development tool, since the work done at the development stage would be reused when verifying the code.

The advantages gained by this technique are liable to be greater for vertical architectures than for horizontal ones. This is because in horizontal architectures instructions are highly compacted and so less likely to be identical. Consider again the Orion microinstruction given above which reloads the instruction register. This instruction also decrements the cache pointer. In other macroinstructions very similar microinstructions are used, except that different tasks are performed at the same time as the instruction register being loaded.

The reuse of collapsed host programs in this manner has not been implemented.

5.5 Removing Unused Variable Assignments

5.5.1 Introduction

One technique which may be used to further simplify the expanded skeleton is the removal of superfluous state variable assignments. Often registers are set by a microinstruction as a side effect of the task actually intended. They are then reset on a subsequent instruction without the earlier value being used. In a high level program a dataflow anomaly such as this would often indicate an error. In a microprogram the situation is common. It often occurs due to the setting of condition flags. Carry out and overflow flags are set as a result of ALU operations. For many microinstructions these values will be unrequired. A further example occurs when pipeline registers are used. These are registers used to store intermediate results which allow operations to be performed in parallel. On each cycle, a client function uses the value in the pipeline register and a server function loads a new value. Frequently the client function does not read the register, but the server still overwrites the value. In the Orion Computer, the cache pipeline register is repeatedly filled with the value in the cache at the location currently addressed. However, it is only used when certain cache microoperations are specified in a microinstruction, so many of the assignments to the corresponding state variable will be redundant.

An expanded skeleton can be simplified greatly if such unused assignments are removed. This is advantageous because it makes the action of each microinstruction clearer, removing the clutter of unimportant machine actions. It also performs some of the work required for proving the correctness of the microcode at an early stage and would speed up simulations.

5.5.2 The Algorithm

Unused state variable assignments are removed from straight line sequences of microcode within the skeleton, which do not contain assertions. When the skeleton is expanded each MI command is replaced by a check-block sequence (CB_i) of the form

CHECK e_i ; BLOCK C_i

Consequently, sequences of code from which unused state variable assignments are removed will have the form

$$CB_1\,;\ CB_2\,;\ldots\,;\ CB_n$$

where each CB_i is a check–block sequence. Sequences of check–blocks of this form are investigated from the front, examining pairs of adjacent check–block sequences and removing assignments from the first of the pair. The set of state variables whose assignments may be removed from the check–block sequence CB_i is given by the set *removables* given by

$$\begin{aligned}&\textit{removables} =\\&\qquad(\textit{state\_variables} - (\textit{used}\ \mathrm{CB}_{i+1})) \cap\\&\qquad(\textit{assigned\_to}\ \mathrm{CB}_i) \cap\\&\qquad(\textit{assigned\_to}\ \mathrm{CB}_{i+1})\end{aligned}$$

where each set is defined below.

- The set *state_variables* contains the set of state variables as declared at the head of the host program.
- The set *used C* contains those state variables which do not appear in a right hand context within the command *C*. This is computed by performing a search of the command and its constituent expressions.
- The set *assigned_to C* contains those state variables assigned to in the command *C*. For a check–block sequence this can easily be determined, since, due to the enforced structure of the expanded host program, all the assignments will be made in a parallel assignment at the end of the block.

Again due to the enforced structure of the expanded host program, all the assignments to be removed will occur in the parallel assignment statement at the end of the block, and so are easily removed. For the same reason only the second check–block sequence need to be examined to determine which variables are used between consecutive assignments.

No unused state variable assignments may be removed from the last check–block pair of the sequence being considered, since it cannot be determined whether the values will be subsequently needed.

The effect of the algorithm is that a state variable is removed if it is assigned to in a microinstruction and re-assigned to with a new value in the next microinstruction without its value being used first. If a variable is not assigned to in the next microinstruction, it is not removed, even if it is assigned to in subsequent microinstructions without being used in the interim. This is because such assignments are more likely to indicate a dataflow anomaly, and so the user ought to be warned instead, though the issuing of warnings was not implemented.

As an example, consider the sequence which occurs in Gordon's computer, given by

```
MI((SW mpc) .= 19); // rir wmar jmp 20
MI((SW mpc) .= 20) // rmem wbuf add jmp21
```

The expanded code is given in Figure 5.2.

In this example, taking the check–block sequence of the first MI instruction to be CB_1 and of the second to be CB_2, the respective sets have the following values:

$$state\_variables = \{\texttt{arg}, \texttt{ir}, \texttt{buf}, \texttt{mar}, \texttt{pc}, \texttt{acc}, \texttt{mem}, \texttt{mpc}\}$$

$$(used\ \mathrm{CB}_2) = \{\texttt{mpc}, \texttt{acc}, \texttt{pc}, \texttt{arg}, \texttt{mem}, \texttt{mar}\}$$

$$(assigned\_to\ \mathrm{CB}_1) = \{\texttt{mar}, \texttt{buf}, \texttt{mpc}\}$$

$$(assigned\_to\ \mathrm{CB}_2) = \{\texttt{buf}, \texttt{mpc}\}$$

$$removables = \{\texttt{buf}\}$$

The assignment to **buf** is the only one which may be removed from the first check–block sequence.

The algorithm could be extended to cross assertions by taking into account which variables the assertion referenced. This would be necessary to prevent incorrect verification conditions being generated. If the system allowed assertions to be changed interactively (this is not currently implemented, though would be useful since producing correct assertions tends to be an iterative task), the skeleton would have to be re-expanded whenever this was done. Consequently only straight line portions of code are considered since the system requires that there be an assertion before each branch.

The method used to model state variable assignment simplifies the process of removing flags. State variables are intended to model flip flops which, if given Z signals on their input wires, leave their state unaltered. As previously described this is modelled explicitly using an assignment of the form

```
    sv := MERGE(sv; new_val)
```

If the new value (*new_val*) does not contain Z signals, the assignment will be rewritten to the form

```
    sv := new_val
```

during the collapsing process. If the new value could contain Z signals the assignment will remain in its original form so a right hand occurrence of the state variable will remain. The previous assignment to it will not be removable. State variables will only be removable if the next value assigned has been guaranteed not to contain Z signals. If the treatment of Z signals had been built into the semantics of the state variable assignment command, unused variable assignment removal would have required analysis to determine whether the value assigned could contain Z signals.

To obtain the full effect of unused variable removal, it ought to be performed repeatedly over the sequences of code until there were no further changes. This is because the only use of the value of a state variable in a microcycle could be an assignment that is removed by the algorithm, allowing the earlier assignment to also be removed. It was not done since situations where it would be useful did not appear to be common for the examples considered. The effect could be more easily achieved by working from the end of the sequence of check–blocks in a single pass,

```
CHECK (SW mpc .= #b_10011);
BLOCK
 $(
    accdisp     == PUT(accdisp; time; SW acc);
    pcdisp      == PUT(pcdisp; time; SW pc);
    idle        == PUT(idle; time; #b_0);
    ready       == PUT(ready; time; #b_0);
    PAR
     $(
         mar    := WS((SW ir)[12..0])
         buf    := WS(#b_000, (SW ir)[12..0])
         mpc    := #s_10100
         time   := INC time
     $)
 $);
CHECK (SW mpc .= #b_10100);
BLOCK
 $(
    accdisp     == PUT(accdisp; time; SW acc);
    pcdisp      == PUT(pcdisp; time; SW pc);
    idle        == PUT(idle; time; #b_0);
    ready       == PUT(ready; time; #b_0);
    PAR
     $(
        buf     := WS(((SW arg) + ((SW mem) !16 (SW mar)))[15..0])
        mpc     := #s_10101
        time    := INC time
     $)
 $)
```

Figure 5.2: A sequence from Gordon's computer

rather than repeatedly from the front. The only overhead would be that the set *assigned_to* would have to be re-evaluated for each check–block sequence after the unused assignments had been removed.

5.5.3 Local Removal

The removal of state variable assignment by the above process could mean that the values of locals were no longer used. The assignments to such locals could also be removed. This is liable to be the case with the associated locals of state variables where their value was not collapsed forwards. The only places where such a local's value is used is in the state variable assignment and check on its well-definedness. The block would have the form

```
BLOCK
 $(
        .
        .
        sv' ==...
        .
        .
        CHECK (WD sv');
        .
        .
        PAR
         $(
            .
            .
            sv := MERGE(sv; sv')
            .
            .
         $)
 $)
```

with no other reference to sv'. If the check had been previously collapsed away, the local assignment would be removable provided that the state variable assignment was unused. The above process has not been implemented.

5.5.4 A Comparison with Other Systems

Clutterbuck [Clutterbuck 86] [Clutterbuck 88] used a similar idea to unused variable assignment removal when converting SPADE-8080 assembly code into FDL, the modelling language for the SPADE verification system. There are some significant differences, however. The FDL model is produced from the SPADE-8080 code using a specially written translation program. Rather than producing FDL code and then scanning it to remove unused assignments in an analogous way to the

SPOOK system, the translator program does not include the assignments in the first place. The rules determining which assignments are unnecessary are built into this program. Since, in the SPADE system, a new translator must be written for each different assembly code considered, the code implementing these rules must be rewritten each time. In the SPOOK system the equivalent code is universal. Furthermore, only flag assignments are considered for non-inclusion in the SPADE system. This contrasts with SPOOK in which any assignment is a possible candidate for removal. This is necessary for microarchitectures since, due to their extra complexity, it is less clear in advance which variables will be set as unwanted side effects. The system itself will determine this separately in each case. In the SPADE system the translator writer must decide which variables constitute flags, and only assignments to those variables can possibly be omitted.

5.6 Combining Successive Microinstructions

When MI commands are sequenced together in the skeleton, information gathered about the values of state variables set in one could be used to further collapse the next. For example, in the sequence from the `Add` module of Gordon's Computer considered previously, executing code at locations 13, 19, 20, 21, 17 and 18 could have been given as

```
MODULE Add
{}
$(
 MI ((SW mpc) .= 13);
 MI #b1;
 MI #b1;
 MI #b1;
 MI #b1;
 MI #b1
$)
{}
```

The collapsing information could be extracted from the expanded versions of the earlier microinstructions and carried forwards into the later ones. For example, the fact that `mpc` has value `#s10011` before the second microinstruction could be noted from the expanded version of the first microinstruction. If taken to its extreme, the collapsing of host programs and moving forwards of state information in this manner corresponds to the symbolic execution of the code.

5.6.1 The Algorithm

Algorithm 5.6.1 is used to collapse state information forwards. As with flag removal, it considers straight line check–block sequences, unbroken by assertions.

$CB_1;\ CB_2; \ldots CB_n$

An association list of the substitutions discovered is maintained as when collapsing host programs, though only substitutions to state variables are held.

Algorithm 5.6.1 (Collapsing State Information Forwards)

```
For i = 1 to number_of_commands do
    1. Substitute values from the a-list into command $CB_i$.
    2. Simplify the command $CB_i$. This involves applying expression and
       command simplifications to the check, and collapsing the block.
    3. Add new substitutions from $CB_i$ to the a-list, removing substitutions
       for any variable for which a new substitution is added. A new
       substitution [c/sv] is formed for each assignment sv := c in the
       parallel assignment of the block where c is a constant.
```

New substitutions are only made from constant assignments. More complex assignments could also be used provided they did not refer to local variables since their value would be restricted to the block. This was not done because large and unreadable expressions can build up in this way. Also, the action of the individual microinstructions would possibly be obscured if this were done.

The assignments from which the substitutions are made are not removed since without analysis of the rest of the code, including annotations, it cannot be ascertained that the value will not be required. It would also obscure the action of each microinstruction. Many assignments of this form may already have been removed at the unused variable removal stage.

In the examples considered in later chapters this process of moving information forwards was not used. Instead all known relevant information was included in the MI command's condition. This has the advantage that the information is made explicit in the skeleton providing useful documentation of the microprogram. Furthermore, the programmer's idea of the values of state variables is actually checked. Even if the information were placed in the MI commands condition, this process could still be useful, however. It would ensure that any information forgotten by the programmer would still be used and also validate that the information in the MI command's condition was correct at an early stage since the process should simplify away the check command resulting from the MI commands condition.

Chapter 6

Verification and Theorem Proving

The proof methods used by the SPOOK system are those of traditional software verification systems. In this chapter, an overview of the general techniques used is given; the rules used to generate verification conditions are described, and the automated help provided by the system to prove these verification conditions is discussed.

6.1 Introduction

6.1.1 Software Verification Techniques

The expanded skeleton is a normal program to which traditional software verification techniques can be applied. Since the SPOOK language is defined by an axiomatic semantics, the method used in the SPOOK system is based on Hoare's method [Hoare 69]. This is automated by a verification condition generator which produces verification conditions from the expanded skeleton. Verification conditions are predicate calculus statements, the truth of which indicates the correctness of the partial correctness specification. In this dissertation, only partial correctness is considered. Termination must be proved separately, though as with partial correctness, software techniques for proving termination will be applicable to the expanded skeleton.

6.1.2 Goals, Tactics and Tacticals

The SPOOK system is embedded in a goal based infrastructure similar to that used for the LCF and HOL systems. The problem at hand is set as a goal. Tactics are then applied to this goal. These are meta-language functions that break a goal into simpler subgoals. It is sufficient to prove all the subgoals to prove the original goal. If a tactic generates no subgoals, then the goal it was applied to is proved. A tactic may also fail if it is not applicable to the goal, in which case the goal is left unaltered.

All outstanding goals are kept on a goal list which may be rotated, allowing the goals to be proved in any order. Tactics may be combined to form more complex tactics using tacticals. These are higher-order meta-language functions which take tactics as arguments and return a new tactic. For example, the tactical `THEN_TRY` applied to two tactics produces a new tactic which applies the first tactic to a goal and then applies the second tactic to all its subgoals, or to the goal itself if the first tactic failed. If any of the subgoals causes failure, then that subgoal is left alone. At any time more than one tactic may be applicable and so tacticals can be used to implement heuristics which determine the tactics to be used. In this way, automatic theorem proving tools can be created.

The tactics and tacticals are written in Standard ML, which also provides the basic user interface to the system. If the tactics available in the system are not powerful enough to prove the verification conditions, new tactics can be written, either encoding additional inference rules or using alternative heuristics. However, the security of the LCF/HOL style systems is not present. Type checking is not used to ensure that only valid theorems are proved.

The system is used in a semi-automatic fashion. The user can apply tactics that largely automate the proof process, or can apply simpler tactics step by step. This removes the burden of dealing with the mass of detail from the user without control having to be completely relinquished. The user can inspect the output at each step before proceeding to the next. In this way errors may be discovered earlier in the proof process, and the user will have a better feel for their location.

6.1.3 Sequents

The theorem proving tools work on sequents. The sequent

$$\Theta \ \texttt{|-}\ P$$

states that conclusion P is true if each of the assumptions in set Θ are true. The notation

$$\texttt{|-}\ P$$

is used as an abbreviation for,

$$\texttt{\{\}}\ \texttt{|-}\ P$$

In the system, sequents, and therefore goals, are represented as a pair consisting of an assumption list and a condition.

6.1.4 Using the System

The main partial correctness specification for the skeleton or that for any of the modules may be set as the initial goal. The first tactic normally to be applied would be `EXPAND_MIS`. This tactic converts all the MI commands to their equivalent collapsed host program form. The resulting goal is the expanded skeleton form of the microcode. It may be visually inspected for errors.

The tactics REMOVE_FLAGS and COMBINE may be used to simplify this goal into a more readable form. The former tactic removes unused assignments from any suitable sequences of code within the partial correctness specification goal as described in Section 5.5. The latter moves state information forwards as described in Section 5.6. Each verification condition generation rule is implemented as a tactic which performs one step of the generation. A general verification condition generation tactic which produces all the verification conditions from a partial correctness specification is formed from these basic tactics.

The AUTO_THM theorem proving tactic described in Section 6.3 can be applied to the verification condition goals. This will remove many of the verification conditions that are straightforward to prove and simplify those that remain, possibly breaking each into several simpler subgoals. The user may indicate which predicate definitions to expand. If the postcondition of the partial correctness specification is given as the conjunction of a series of predicates—a fairly natural way to structure the specification—then each conjunct will ultimately correspond to separate subgoals. For example, a postcondition

```
{AND(...) /\ CACHE_OK(...)}
```

would cause goals of the following form to be generated

```
... |- AND(...)
... |- CACHE_OK(...)
```

If these predicates are not initially expanded, they may be proved separately. Therefore, if they cannot be proved due to an error in the microcode or specifications, the location of the error will be much easier to determine. Furthermore, it will be much more obvious which of the predicates in the postcondition have been validated and which have not, when the system cannot prove all the verification conditions. Avoiding expanding predicate definitions in the assumptions is also useful if the predicates are not required for the proof of a particular subgoal.

In general, the AUTO_THM tactic will not be powerful enough to prove all the subgoals generated. The user may then apply any of the basic theorem proving tactics of the system to prove these goals. If they can still not be proved, new tactics could be introduced. However, as previously noted, the security of the LCF and HOL theorem proving systems is not present in SPOOK. This means that erroneous tactics could apparently prove goals which were actually untrue. There is a danger, if a tactic is written with a particular goal in mind, that insufficient care is taken to ensure its correctness. For this reason, it is possibly more advisable to prove unproved goals outside the SPOOK system, using theorem proving tools such as HOL.

6.2 Verification Condition Generation

Rules derived from the axiomatic semantics presented in Appendix A are used to produce verification conditions for a given partial correctness specification. They are encoded in the system as tactics. A rule

$$\frac{S_1 \ldots S_n}{G}$$

would be implemented as a tactic which would form subgoals S_i from goal G. Verification condition generation using tactics in this way was adopted by Gordon [Gordon 88].

6.2.1 The WP Operator

A predicate transformer WP is used in the formulation of many of the verification condition generation rules. The condition WP(C,Q) is such that after executing command C, from a state in which that condition is true, results in a state satisfying condition Q. It is the weakest such condition in that it is implied by all other conditions satisfying the above. It is not used as a calculus in which to define the semantics of the language in the way others have used such operators, since the formal definition of the language is given by the Hoare rules of Appendix A. Instead it is used in the verification condition generation rules as a convenient meta-notation for describing various conditions. For example, in the general verification condition generation Rule VCGEN.1 described later, the use of WP allows a single rule to be used for several different commands. It is also used in one of the sequencing rules, VCGEN.3, to effectively compute a suitable intermediate annotation when the user did not provide one. The meta-function WP corresponds directly with an ML function *wp* used in the implementation of the verification condition generation rules. WP is defined recursively, though is not defined for all commands since this is not required in the formulation of the verification condition generation rules.

$$\begin{array}{ll}
\mathsf{WP}(\texttt{SKIP},\ Q) & = Q \\
\mathsf{WP}(\texttt{ABORT},\ Q) & = \texttt{F} \\
\mathsf{WP}(v\ \texttt{==}\ e,\ Q) & = Q[e/v] \\
\mathsf{WP}(\texttt{CHECK}\ e,\ Q) & = Q\ \texttt{/\textbackslash}\ \texttt{IS\_ONEP}\ e \\
\mathsf{WP}(\texttt{PAR \$(}\ v_1\ \texttt{:=}\ e_1 \ldots v_n\ \texttt{:=}\ e_n\texttt{\$)},\ Q) & = Q[e_1/v_1,\ldots,v_n/e_n] \\
\mathsf{WP}(C_1\ \texttt{;}\ C_2,\ Q) & = \mathsf{WP}(C_1,\ \mathsf{WP}(C_2,\ Q)) \\
\mathsf{WP}(\texttt{BLOCK}\ C,\ Q) & = (\mathsf{WP}(C,\ Q[U/\ locals]))[U/locals]
\end{array}$$

The notation $Q[e/v]$ is used to denote the condition Q with all occurrences of variable v replaced by expression e. The notation $Q[e_1/v_1,\ e_2/v_2,\ldots,\ e_n/v_n]$ denotes the parallel substitution of the expressions e_i for the corresponding v_i. The abbreviation $Q[U/locals]$ is used to denote the condition Q with all occurrences of all the local variables replaced by suitably sized constants consisting of U values.

In order that the verification condition generation rules be consistent with the Hoare axioms, the definition of WP must satisfy the property

$$\texttt{|-}\{\ \mathsf{WP}(C,\ Q)\}\ C\ \{\ Q\ \}$$

for each command upon which it is defined. This may be proven fairly straightforwardly from the definition and Hoare axioms by structural induction over the commands.

6.2.2 General Commands

The following rule is used to generate verification conditions for many of the commands.

VCGEN.1 (General)

$$\frac{\vdash P \texttt{==>} \mathsf{WP}(C, Q)}{\vdash \{P\}\ C\ \{Q\}}$$

where C is one of `SKIP`, `ABORT`, `CHECK`, assignment, `PAR` or `BLOCK`.

This states that the verification condition generated for $\{P\}$ C $\{Q\}$ for an appropriate command C, is P ==> WP(C, Q). For example, the verification condition for the partial correctness specification $\{P\}$ `SKIP` $\{Q\}$ is P ==> Q. The rule follows from the property proven in the previous section about WP, together with precondition strengthening (HOARE.14).

$$\frac{\vdash P \texttt{==>} \mathsf{WP}(C, Q) \quad \vdash \{\mathsf{WP}(C, Q)\}\ C\ \{Q\}}{\vdash \{P\}\ C\ \{Q\}}$$

where C is one of `SKIP`, `ABORT`, `CHECK`, assignment, `PAR` or `BLOCK`.

6.2.3 Sequencing

Two rules are used for sequences. The first rule is used whenever an assertion is placed between the commands. When the second command in the sequence is a conditional command, a loop command or a CALL_MODULE command, an annotation must be given. Annotations may also optionally be given when the second command is not one of the above.

VCGEN.2 (Annotated Sequencing)

$$\frac{\vdash \{P\}\ C_1\ \{R\} \qquad \vdash \{R\}\ C_2\ \{Q\}}{\vdash \{P\}\ C_1\ ;\{R\}\ C_2\ \{Q\}}$$

This rule states that the verification conditions generated for $\{P\}$ C_1 ;$\{R\}$ C_2 $\{Q\}$ are those generated for $\{P\}$ C_1 $\{R\}$ together with those for $\{R\}$ C_2 $\{Q\}$. It follows directly from the sequencing axiom (HOARE.3), where the annotation gives the condition which holds after execution of the first command.

If no annotation is given then the second sequencing rule is used.

VCGEN.3 (Sequencing)

$$\frac{\vdash \{P\}\ C_1\ \{\ \mathsf{WP}(C_2,\ Q)\}}{\vdash \{P\}\ C_1\ ;\ C_2\ \{Q\}}$$

Here the WP operator effectively computes a suitable intermediate assertion. The rule follows from the WP property and the sequencing axiom (HOARE.3).

$$\frac{\vdash \{P\}\ C_1\ \{\mathsf{WP}(C_2,\ Q)\} \qquad \vdash \{\ \mathsf{WP}(C_2,\ Q)\}\ C_2\ \{Q\}}{\vdash \{P\}\ C_1\ ;\ C_2\ \{Q\}}$$

It is assumed for this rule that the second command does not contain subcommands containing annotations. This must be so as otherwise the annotations would be ignored when computing WP, and not checked to be consistent with the program. Without loss of generality, it can be assumed that the second command is not a sequencing operator. The only other compound command possible is the block command. This is assured not to require annotations within an expanded skeleton due to the structure of host programs. Note that MI commands are also excluded from the analysis since they are replaced by collapsed host programs before the verification condition generation stage.

6.2.4 CALL_MODULE

The rule used for CALL_MODULE follows from the precondition strengthening rule (HOARE.14), the postcondition weakening rule (HOARE.13) and the module calling axiom (HOARE.9).

VCGEN.4 (Calling Modules)

$$\frac{\begin{array}{l}\texttt{|-}\ P\ \texttt{==>}\ P_m\\ \texttt{|-}\ Q_m\ \texttt{==>}\ Q\\ \texttt{|-}\ \{P_m\}\ c_m\ \{Q_m\}\end{array}}{\texttt{|-}\ \{P\}\ \texttt{CALL\_MODULE}\ m\ \{Q\}}$$

where P_m and Q_m are the preconditions and postconditions of module m and c_m is the body of the module.

6.2.5 IF

The rule for IF follows from the If axiom (HOARE.10) and precondition strengthening (HOARE.14).

VCGEN.5 (If)

$$\frac{\begin{array}{l}\texttt{|-}\ \{P\ \texttt{/\textbackslash}\ \texttt{IS\_ONEP}(b_1)\}\ C_1\ \{Q\}\\ \quad\quad\quad\quad\vdots\\ \texttt{|-}\ \{P\ \texttt{/\textbackslash}\ \mathrm{IS\_ONEP}(b_n)\}\ C_n\ \{Q\}\\ \texttt{|-}\ \{P\ \texttt{/\textbackslash}\ \mathrm{IS\_ZEROP}(b_1 \ldots b_n)\}\ C_{n+1}\ \{Q\}\\ \texttt{|-}\ P\ \texttt{==>}\ \mathrm{BINARYP}(b_1 \ldots b_n)\\ \texttt{|-}\ P\ \texttt{==>}\ \mathrm{EXCLUSIVEP}(b_1 \ldots b_n)\end{array}}{\texttt{|-}\ \{P\}\ \mathrm{IF}\ \texttt{||}\ b_1\ \texttt{=>}\ C_1\ \texttt{||} \ldots b_n\ \texttt{=>}\ C_n\ \mathrm{DEFAULT}\ C_{n+1}\ \{Q\}}$$

6.2.6 CASE

The CASE rule similarly follows from the Case axiom (HOARE.11) and precondition strengthening (HOARE.14).

VCGEN.6 (Case)

$$\frac{\begin{array}{l} \texttt{|-}\ \{P\ \texttt{/\\}\ e\ \texttt{=}\ b_1\}\ C_1\ \{Q\} \\ \quad \cdot \\ \quad \cdot \\ \texttt{|-}\ \{P\ \texttt{/\\}\ e\ \texttt{=}\ b_n\}\ C_n\ \{Q\} \\ \texttt{|-}\ \{P\ \texttt{/\\}\ \texttt{\~{}}e\ \texttt{=}\ b_1\ \texttt{/\\..\~{}e}\ \texttt{=}\ b_n\}\ C_{n+1}\ \{Q\} \\ \texttt{|-}\ P\ \texttt{==>}\ \texttt{BINARYP}(e) \end{array}}{\texttt{|-}\ \{P\}\ \texttt{CASE}\ e\ \texttt{||}\ b_1\ \texttt{=>}\ C_1\ \texttt{||..}b_n\ \texttt{=>}\ C_n\ \texttt{DEFAULT}\ C_{n+1}\ \{Q\}}$$

6.2.7 WHILE

The rule for while loops follows from the While axiom (HOARE.12), precondition strengthening (HOARE.14) and postcondition weakening (HOARE.13).

VCGEN.7 (While)

$$\frac{\begin{array}{l} \texttt{|-}\ \{R\ \texttt{/\\}\ \texttt{IS\_ONEP}(b)\}\ C\ \{R\ \texttt{/\\}\ \texttt{BINARYP}(b)\} \\ \texttt{|-}\ R\ \texttt{/\\}\ \texttt{IS\_ZEROP}(b)\ \texttt{==>}\ Q \\ \texttt{|-}\ P\ \texttt{==>}\ R\ \texttt{/\\}\ \texttt{BINARYP}(b) \end{array}}{\texttt{|-}\ \{P\}\ \texttt{WHILE}\ b\ \texttt{DO}\ \{R\}\ C\ \{Q\}}$$

6.3 Proving Verification Conditions

The SPOOK system includes very simple theorem proving tools with which the verification conditions can be proved. These are based on several inference rules, together with a simple heuristic for applying them. The collection of rules is fairly small and ad hoc. However, the majority of the verification conditions generated are of a fairly simple nature, and so the rules used are often sufficient. Any that cannot be proven, either due to insufficient rules or due to the heuristics used for applying them being inadequate, are left to the user to prove by other methods. For example, if a suitable HOL theory was developed, the HOL system could be used as a tool to aid an expert in proving the remaining verification conditions.

The automated theorem proving tactic, AUTO_THM, uses tacticals to combine the various basic tactics implemented in the system. Each stage generates new, though possibly unchanged, subgoals from the goals of the previous stage. The tactics are applied to all of the subgoals generated from the previous stage. Below is an overview of this tactic. The details of the individual steps are described in the remainder of this section.

The initial verification condition will have the form

$$\texttt{|-}\ P_1\ \texttt{/\\...\\/}\ P_n\ \texttt{==>}\ Q_1\ \texttt{/\\.../\\}\ Q_m$$

This is first split into a series of goals with individual assumptions.

$\{P_1,\ldots,P_n\}$ |- Q_1

.

.

.

$\{P_1,\ldots,P_n\}$ |- Q_m

The resulting goals may be trivially seen to be true, such as when the conclusion is identical to one of the assumptions. The goals are compared with several such 'simple rules' and any that are matched are removed from the goal list. Next equality information held in the assumptions is used. If a goal has an assumption E_1 = E_2, then occurrences of expression E_1 may be replaced by E_2. Similarly if an assumption asserts the truth of a predicate then occurrences of that predicate in the conclusion may be replaced by T. The conditions and expressions within the resulting goal may now be simplifiable, and so rewrite rules are applied. This could have converted many of the goals to a trivially true form and so the simple rules are retried. Finally, the predicate definitions suggested by the user are expanded in the remaining goals and the above process repeated once. Examples of this process are given in subsequent chapters.

6.3.1 Splitting Verification Conditions

The initial verification condition will have the form

|- P_1 /\.../\ P_n ==> Q_1 /\.../\ Q_m

The tactics `DISCH` and `ASSUM_SPLIT_CONJ` are used to split the antecedent into individual assumptions.

`DISCH`

Antecedents of implications may be moved to the assumptions.

$$\frac{\{P,\ldots\} \mid\!- Q}{\{\ldots\} \mid\!- P \Longrightarrow Q}$$

`ASSUM-SPLIT-CONJ`

Conjuncts in the assumptions may be split into independent assumptions.

$$\frac{\{\ldots, P, Q,\ldots\} \mid\!- R}{\{\ldots, P \wedge Q,\ldots\} \mid\!- R}$$

`SPLIT_CONJ` is then used to split the consequent, allowing each conjunct to be proven separately.

`SPLIT-CONJ`

Conjuncts in the consequent may be proven independently.

$$\frac{\Theta \mid\!- P \quad \Theta \mid\!- Q}{\Theta \mid\!- P \wedge Q}$$

A verification condition

$$\texttt{|-}\ P_1\ \texttt{/\textbackslash}\ldots\texttt{/\textbackslash}\ P_n\ \texttt{==>}\ Q_1\ \texttt{/\textbackslash}\ldots\texttt{/\textbackslash}\ Q_m$$

is converted to the goals

$$\{P_1,\ldots,P_n\}\ \texttt{|-}\ Q_1$$
$$\vdots$$
$$\{P_1,\ldots,P_n\}\ \texttt{|-}\ Q_m$$

This is performed automatically when generating the verification conditions.

6.3.2 Simple Rules

The next stage is to attempt to match each of the subgoals to one of several simple rules. These are templates of simple goals that often occur in practice. Any goals that can be matched are considered to be proved and generate no subgoals. Any that cannot be so proved are left unchanged. Applying the simple rules immediately ensures that effort is not wasted applying more complex rules to trivial goals. Since many verification conditions are of a very simple nature, it is often worthwhile trying very simple things first.

`P-ENTAILS-P`
: Any condition P is true if it is an assumption.

$$\{\ldots,P,\ldots\}\ \texttt{|-}\ P$$

`ALL-ENTAILS-T`
: T is true, whatever the assumptions.

$$\{\ldots\}\ \texttt{|-}\ \texttt{T}$$

`F-ENTAILS-P`
: Under a false assumption, anything is true. (Goals of this form will be associated with the verification conditions from unexecutable paths.)

$$\{\ldots,\texttt{F},\ldots\}\ \texttt{|-}\ P$$

`E-EQ-E`
: An equality between two identical expressions is true under any assumptions.

$$\{\ldots\}\ \texttt{|-}\ E\ \texttt{=}\ E$$

6.3.3 Substitutions

If the goals are not of this very simple form, information from the assumptions is used to substitute for known values in the consequent. This is done using the tactics EQSUB and PREDSUB.

EQSUB

If it is assumed that two expressions are equal, then occurrences of the first may be replaced by occurrences of the second within the conclusion.

$$\frac{\{\ldots,\ E_1\ =\ E_2,\ldots\}\ \ \texttt{|-}\ldots E_2\ldots}{\{\ldots,\ E_1\ =\ E_2,\ldots\}\ \ \texttt{|-}\ldots E_1\ldots}$$

EQSUB assumes that the equality is given in a form such that the first expression is the most useful one to be substituted for. This will not always be so. An equivalent rule making the substitution in the opposite direction is not given to avoid the possibility of infinite looping when theorem proving. Since the assumptions are largely generated from the annotations, they can often be written in the most useful form. For example,

```
Op = Mem !32 Pc
```

is more likely to be useful than

```
Mem !32 Pc = Op
```

PREDSUB

If a predicate is assumed true, then occurrences of it in the consequent may be replaced by T.

$$\frac{\{\ldots,P,\ldots\}\ \ \texttt{|-}\ldots\texttt{T}\ldots}{\{\ldots,P,\ldots\}\ \ \texttt{|-}\ldots P\ldots}$$

where P is a predicate

6.3.4 Expression Simplification

By substituting for known values from the assumptions, the expressions within the consequent may become simplifiable. The tactic EVAL_CONSTS is first used to replace constant subexpressions by their value. EXP_REWRITE is then used to apply the expression rewrite rules, simplifying the goals in a similar way to that used when simplifying commands during the collapsing process. These simplifications are applied to both the conclusion and assumptions of the goal. The set of rewrite rules used is given in Appendix C.

Before simplifying the expressions, EXP_REWRITE collects information from the assumptions concerning whether variables could possibly hold X, Z or U signals. Two predicates assert this kind of information: BINARYP and INPUT_BINARYP. The former asserts that its single word argument will only have a value consisting of 0 and 1 signals. The latter gives the same information for its history argument. Thus, if an assumption

```
BINARYP(SW v)
```

exists for a state variable *v* it is noted that *v* will never return X values. Since *v* is a state variable, it is already known that it cannot return U or Z values. Similarly if an assumption

```
INPUT_BINARYP(h)
```

exists, for an input variable, *h*, it is noted that *h* will at no time return X or Z signals. Since *h* is a history variable it will never return U signals. This information is used in the simplification process in the same way that the corresponding information was used in the collapsing process.

The same set of rewrite rules are used as when collapsing the host program, though some are of more use during theorem proving. In particular, the rules about expressions combining selection, merge and subscription update operators are mainly used during theorem proving. These match expressions of the form

```
(SW MERGE(a; c != v))[n..m]
```

where *c* is a constant. Expressions having this structure will arise frequently in verification conditions for production machines such as the Orion. The expression

```
MERGE(a; c != v)
```

updates a memory structure *a* at address *c* with value *v*. Converting the result to a word and then making a selection will access the updated memory structure. The selection will often have been rewritten from a subscript expression,

```
(SW MERGE(a ; c != v)) !k d
```

If the addresses *c* and *d* are the same then the expression will simplify to *v*—the new value of that location. If they are definitely different then the expression would simplify to

```
(SW a) !k d
```

or, returning to the original selection form,

```
(SW a)[n..m]
```

since the updated location would be irrelevant.

It is important that such expressions are simplified since otherwise voluminous expressions can result, especially when multiple updates to a memory are performed. In the Orion computer host program, the register bank is represented as a single memory (**res**) so updates to different registers within a program all update this one variable.

This situation is very similar to that associated with the more familiar array update. In assembler level and high level systems it is avoided by giving the different words within a memory symbolic names. However, this just introduces a new problem of aliasing—ensuring that different names refer to different locations.

6.3.5 Condition Simplification

Condition rewrite rules may now be applicable. For example, a goal

```
{P /\ T} |- Q \/ F
```

could be rewritten to

```
{P} |- Q
```

The condition rewrite rules are similar in nature to expression rewrite rules, except that they apply to predicate calculus conditions rather than to expressions. As with expression rewriting they are applied to all subconditions, not just the leading condition. The set of condition rewrite rules used are given in Appendix C.

Many expressions will have been simplified to a constant by the expression simplification stage. Therefore, many of the condition rewrite rules used convert a predicate applied to a constant to either `T` or `F`, effectively evaluating the predicate. Since conditions are predicate calculus formulae, they are not in general executable and so rewrite rules are used to do this evaluation, rather than using an evaluator tactic such as `EVAL_CONSTS`. For example, the rule, COND.2(Binaryp_T), converts conditions of the form

```
BINARYP(c)
```

where c is a constant word containing only the signals 1 and 0, to `T`. Other rules then simplify conditions containing `T` and `F`.

In addition, the rule, COND.13(E_eq_E) replaces an equality between identical expressions by `T`, and COND.14(Is_onep_wd) performs well-defined analysis on expressions. This is useful when attempting to prove the verification conditions generated from well-defined checks.

6.3.6 Rematching with the Simple Rules

After the above simplifications have been performed, the goals produced are rematched with the simple rules, and any that do match are removed. Since the condition rewrite rules, on the whole, convert conditions to `T` or `F`, the simple rules `ALL_ENTAILS_T` and `F_ENTAILS_P` are likely to match many goals.

6.3.7 Expanding Predicate Definitions

The next step, performed on any goals that remain, is to expand the predicate definitions within them. SPOOK allows the definitions of predicates to be given as part of the program. These definitions may be expanded, replacing occurrences of the predicate by its body with the formal parameters replaced by the actual parameters. Definitions may be expanded within the assumptions, the conclusion or both. This is done using the tactic `REWRITE_WITH_DEFN_LIST`.

This tactic is not used earlier in the proof process because expanding definitions involves increasing the size of the goals dramatically. As much simplification as possible should be done initially to reduce this effect as much as possible. Often goals of the form

P `|-` P

where P is a predicate, arise. Goals of this form can easily be proved by matching with the simple rule `P_ENTAILS_P`. If the predicates were expanded immediately, this matching would involve far more work. Moreover, many of the predicates used have definitions that are a series of conjuncts. Therefore, a goal of the above form could be split into a series of subgoals after expanding the definition, requiring many matchings with the simple rule. Rather than attempting to expand all the definitions within the goals, the user supplies the names of those to be expanded, thus affording them some control. Once the predicates have been expanded, the above process is repeated on the new goals. If any subgoals are left after this stage, the `AUTO_THM` tactic gives up, leaving them for the user to prove.

Chapter 7

Gordon's Computer

In this chapter, a case study of the verification of the microprogram of Gordon's Computer is described, detailing each of the steps involved. Statistics illustrating the success of the system are discussed, together with an analysis of the errors discovered. This example, being fairly simple, provides a good illustration of the techniques used by the system. In particular it shows how a non-modular, unstructured microprogram can be verified as a series of structured and independent modules.

The SPOOK system was used to perform a reverification of a very simple microprogrammed computer—Gordon's computer. Gordon's computer was designed as a hardware verification test case. It provides many of the features of a real machine, whilst still being simple enough for the verification task to be tractable. Consequently, several systems have been used to verify it at various levels. [Gordon 81b] [Joyce 86] [Barrow 84b] [Davie 88].

The host program, describing the microarchitecture of Gordon's computer, was converted to the SPOOK language from the BSPL specification [Richards 86]. The memory image version of the microprogram was included in this host program. A skeleton giving a modular, hierarchical version of the flow graph of the microprogram, which itself was neither hierarchical nor modular, was also written. This was structured so that each macroinstruction of the architecture of Gordon's computer was self contained in a module. This allowed each macroinstruction to be separately specified and verified. The target specification consisted of a precondition and postcondition for the main body of the skeleton, together with those for each module. Additional annotations were also placed within the skeleton to aid the proof. During the verification process, an error was discovered in the original version of the microprogram being verified due to an unprovable verification condition. Other errors were also found in the host program, target specification, skeleton and annotations in the course of the verification.

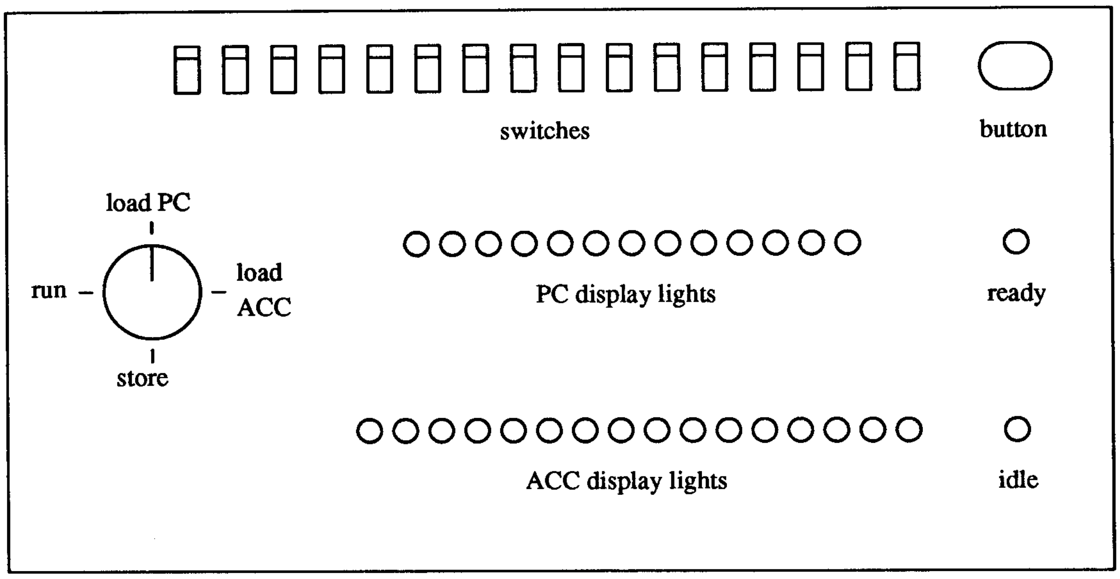

Figure 7.1: The front panel of Gordon's computer

7.1 The Target Level

The target level of Gordon' computer contains two registers: a 16 bit accumulator and a 13 bit program counter. Memory consists of 16 bit words in a 13 bit address space. The front panel is shown in Figure 7.1. It consists of an idle light, a run light, thirteen lights displaying the contents of the program counter and a further sixteen lights displaying the accumulator contents. There is also a button, a four position knob and sixteen binary switches. Gordon's computer has two execution modes—an idle mode in which the data can be loaded into the registers and memory from the switches, and a run mode in which instructions in memory are executed. In both modes the ready light is lit at the start of each macrocycle.

7.1.1 Idle Mode

When idling, the idle light is lit and the computer does nothing. If the button is pressed whilst the computer is idling, the computer obeys one of four instructions dependent upon the knob position.

LOAD_PC : knob = 0

Load the values given by the thirteen low order switches into the program counter, then continue to idle.

LOAD_ACC : knob = 1

Load the values given by the switches into the accumulator, then continue to idle.

LOAD_MEM : knob = 2

Load the value in the accumulator into the memory position addressed by the program counter, then continue to idle.

RUN : knob = 3

Switch to run mode (described below) and start executing the program held in memory which begins at the location addressed by the program counter.

7.1.2 Run Mode

When in run mode, instructions are repeatedly fetched from memory at the position addressed by the program counter. Instructions have two parts: a 3 bit opcode and a 13 bit address. Eight different instructions are available.

HLT : opcode = 0

Stop instruction execution and return to the idle mode. Leave the program counter unaltered.

JMP : opcode = 1

Load the program counter with the address part of the instruction (i.e., jump to that address).

JZE : opcode = 2

Test the value of the accumulator. If it is zero, load the program counter with the address given in the address field of the instruction, otherwise increment the program counter.

ADD : opcode = 3

Add the contents of the accumulator to the value in memory at the given address, leaving the result in the accumulator. Increment the program counter.

SUB : opcode = 4

Subtract the value in memory at the given address from the contents of the accumulator, leaving the result in the accumulator. Increment the program counter.

LDA : opcode = 5

Load the accumulator with the value at the given address in memory. Increment the program counter.

STA : opcode = 6

Store the accumulator's value to the given address in memory. Increment the program counter.

SKP : opcode = 7

Increment the program counter.

In addition to executing the HLT instruction, the computer can be returned to idle mode by pressing the button. If this is done, the computer will idle after the current instruction has been executed.

7.2 The Host Level

In this section I describe the host machine and corresponding source host program upon which Gordon's computer is implemented. The microarchitecture is fairly simple. The block diagram is given in Figure 7.2. In addition to the registers, inputs and outputs corresponding to the target level resources extra registers are provided.

- `arg`—a 16 bit register used to hold arguments to the arithmetic and logic unit (ALU).
- `ir`—a 16 bit instruction register.
- `buf`—a 16 bit register which stores ALU results.
- `mar`—a 13 bit machine address register.
- `mpc`—a 5 bit microprogram counter.

The source host program declarations are given below.

```
INPUT           button   : 1
                knob     : 2
                switches : 16

OUTPUT          pcdisp   : 13
                accdisp  : 16
                ready    : 1
                idle     : 1

STATE_VARIABLES
                arg      : 16
                ir       : 16
                buf      : 16
                mar      : 13
                pc       : 13
                acc      : 16
                mem      : 13 -> 16
                mpc      : 5
```

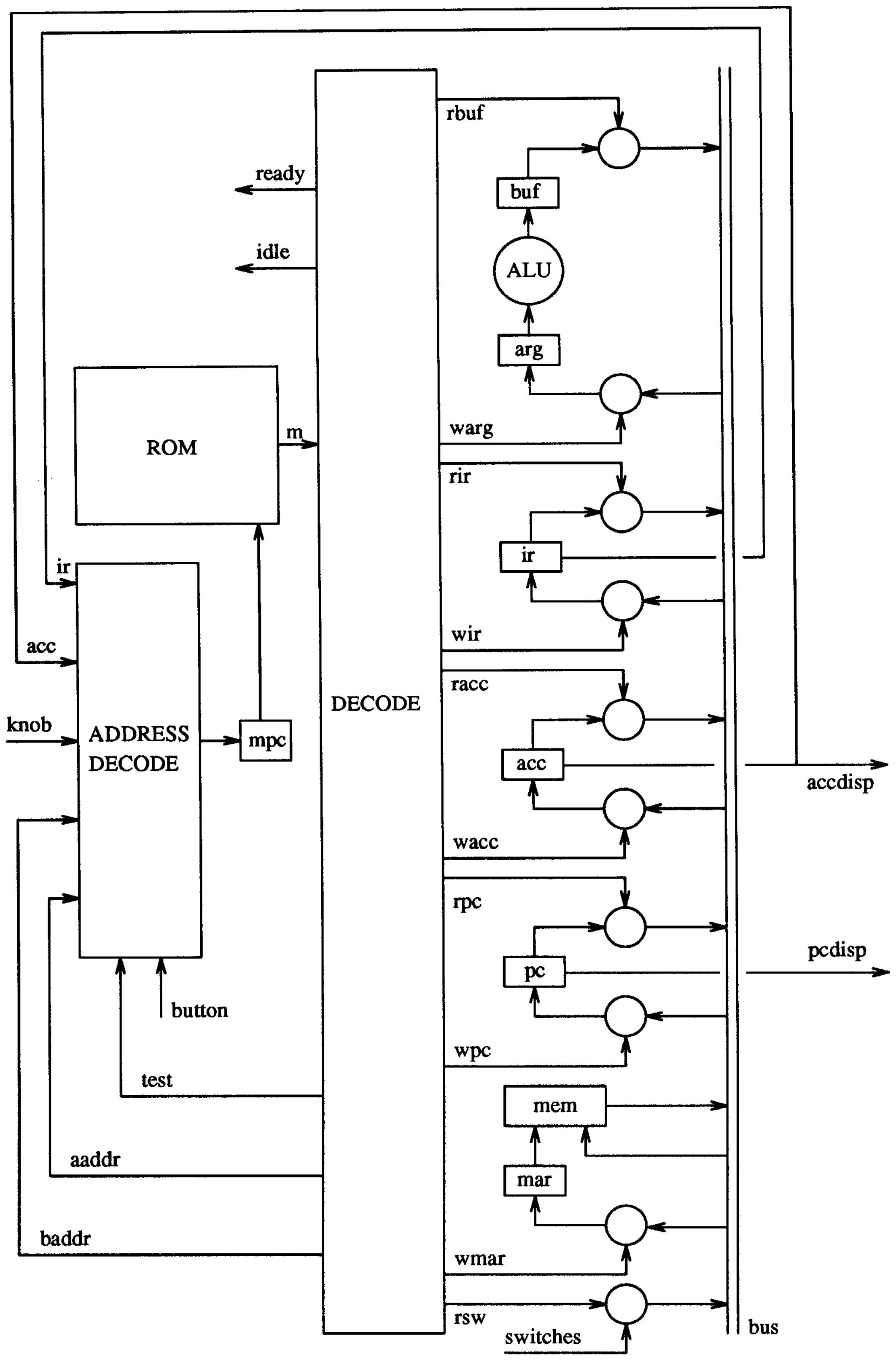

Figure 7.2: The block diagram for an implementation of Gordon's computer

The control unit consists of a read only microprogram memory, instruction fetch and decoding logic, and control signals. The read only memory is described in the source host program by assigning a constant representing the memory contents to a local, *rom.*

```
rom == #b_00_00_000000_0_00_000_00000_00000_000,...
```

This models the read only nature of the memory, though has the disadvantage that the actual microprogram becomes an integral part of the host program. An alternative method would be to use a state variable, but give the host program no facilities to change it. The concatenation operator and underscores within the constant can be used to break it into words and fields making it more readable. The full constant is given in Appendix D. The assembler version is given in Figure 7.3. The code is the same as that used in previous verification attempts. The first column of the assembler notation indicates the address of the microinstruction, the second and third indicate the source and destination registers respectively, the next indicates the ALU function, if any, to be executed, and the final column indicates the next location to jump to.

The store contains 32 horizontal control words, each of 29 bits. This is a departure from the control store described in earlier proofs which consisted of 30 bit control words where the top bit was not used. The address of the next microword to be decoded is held in the microprogram counter, `mpc`. In the source host program, the fetched word is stored in a local, `m`.

```
m == rom !29 mpc
```

The various control signals are decoded using the selection operator.

```
rsw     == m[28]
wmar    == m[27]
memcntl == m[26..25]
wpc     == m[24]
rpc     == m[23]
wacc    == m[22]
racc    == m[21]
wir     == m[20]
rir     == m[19]
warg    == m[18]
alucntl == m[17..16]
rbuf    == m[15]
ready   == m[14]
idle    == m[13]
aaddr   == m[12..8]
baddr   == m[7..3]
test    == m[2..0]
```

Location	Source	Destination	Function	Next Location
0	`ready`	`idle`		`jbut 0 1`
1				`knob 1`
2	`rsw`	`wpc`		`jmp 0`
3	`rsw`	`wacc`		`jmp 0`
4	`rpc`	`wmar`		`jmp 7`
5	`ready`			`jbut 6 0`
6	`rpc`	`wmar`		`jmp 8`
7	`racc`	`wmem`		`jmp 0`
8	`rmem`	`wir`		`jmp 9`
9				`jop 10`
10				`jmp 0`
11	`rir`	`wpc`		`jmp 5`
12				`jze 17 11`
13	`racc`	`warg`		`jmp 19`
14	`racc`	`warg`		`jmp 22`
15	`rir`	`wmar`		`jmp 24`
16	`rir`	`wmar`		`jmp 25`
17	`rpc`	`wbuf`	`inc`	`jmp 18`
18	`rbuf`	`wpc`		`jmp 5`
19	`rir`	`wmar`		`jmp 20`
20	`rmem`	`wbuf`	`add`	`jmp 21`
21	`rbuf`	`wacc`		`jmp 17`
22	`rir`	`wmar`		`jmp 23`
23	`rmem`	`wbuf`	`sub`	`jmp 21`
24	`rmem`	`wacc`		`jmp 17`
25	`racc`	`wmem`		`jmp 17`
26				`jmp 0`
27				`jmp 0`
28				`jmp 0`
29				`jmp 0`
30				`jmp 0`
31				`jmp 0`

Figure 7.3: The assembly code for the microprogram of Gordon's computer

The least significant control fields of the microinstruction concern the microsequencing. The first field determines which microsequencing operation is performed and the next two give possible next addresses. Ready and idle fields indicate whether the corresponding lights should be switched on for that microcycle. The remaining signals indicate whether particular registers should be used as sources or destinations and which, if any, memory and ALU operations should be performed.

The accumulator and program counter lights display the value currently stored in the respective register, as set at the end of the previous cycle.

```
accdisp == acc
pcdisp  == pc
```

The registers are connected by a 16 bit bus. This can take its value from several sources: the switches, program counter, accumulator, instruction register, buffer, or memory as addressed by the memory address register. The source selected depends on the control signals `rsw`, `rpc`, `racc`, `rir`, `rbuf` and `memcntl` respectively.

```
bus == GUARDED
          || rsw     .= 1    => switches
          || rpc     .= 1    => #b000,pc
          || racc    .= 1    => acc
          || rir     .= 1    => #b000,ir[12..0]
          || rbuf    .= 1    => buf
          || memcntl .= 1    => mem !16 mar
```

The program counter (pc) and the part of the instruction register selected (`ir[12..0]`) are only 13 bits long and so are zero padded before being put on the 16 bit bus.

The value on the bus can be stored directly in any of the registers, other than the microprogram counter and the buffer. (It can be placed in the latter via the ALU.) It can also be written to the memory location as addressed by the memory address register.

```
mem := SWITCH  memcntl || 2  => mar != bus
acc := SWITCH  wacc    || 1  => bus
ir  := SWITCH  wir     || 1  => bus
arg := SWITCH  warg    || 1  => bus
mar := SWITCH  wmar    || 1  => bus
pc  := SWITCH  wpc     || 1  => bus
```

The variable `bus` will be automatically truncated when assigned to sinks shorter than itself. This occurs in the assignments to `mar` and `pc`.

The buffer register stores the result of the ALU operation. This can be the value on the bus; it incremented; or it added to or subtracted from the argument register contents. The function is determined by the `alucntl` field in the microword.

```
buf := SWITCH alucntl
        || 0  =>        bus
        || 1  => 1   + bus
        || 2  => arg + bus
        || 3  => arg - bus
```

The test control field determines the next address to be loaded into the microprogram counter. Two microword fields, aaddr and baddr are possible sources. If test is 0 the source is aaddr. If it is 1, the choice is dependent on the current value of the button, and if 2, on whether the accumulator is 0 or not. In the last two modes, the value of aaddr is used, but modified with the value of the knob or part of the instruction register, respectively.

```
mpc := SWITCH test
        || 0   => aaddr
        || 1   => (SWITCH button
                     || 1 => baddr
                     DEFAULT aaddr)
        || 2   => (SWITCH acc
                     || 0 => baddr
                     DEFAULT aaddr)
        || 3   => aaddr + knob
        || 4   => aaddr + ir[15..13]
```

7.3 The Expanded Host Program

The first task when expanding the source host program is to deduce the types of the locals and add their declarations. This is fairly straightforward. The locals are those variables which are not otherwise declared, together with an associated local for each state variable and output. The lengths are determined by examining the right hand sides of each of the assignments in turn.

By the concatenation length rule, the length of rom is the sum of the lengths of the constants which are concatenated together.

```
rom : 29 -> 32
```

The right hand side of the assignment to m is a subscription, so the length is implicit in the operator.

```
m :  29
```

Each of the locals representing the fields of the microword take their value from a selection, so the length is the number of signals selected.

```
rsw     :  1
wmar    :  1
memcntl :  2
```

```
wpc     :  1
rpc     :  1
wacc    :  1
racc    :  1
wir     :  1
rir     :  1
warg    :  1
alucntl :  2
rbuf    :  1
aaddr   :  5
baddr   :  5
test    :  3
```

The outputs `accdisp`, `pcdisp`, `ready` and `idle` have associated locals which inherit their length.

```
accdisp' :  16
pcdisp'  :  13
ready'   :  1
idle'    :  1
```

The local `bus` takes its value from a guarded expression so has the same length as the shortest of the possible results. The others will be truncated to this length. In this case all the possibilities have length 16.

```
bus :  16
```

Each of the locals associated with the state variables inherit the respective lengths of the latter.

```
mem' :  16
acc' :  16
ir'  :  16
arg' :  16
mar' :  13
pc'  :  13
buf' :  16
mpc' :  5
```

The next stage is to place the body of the program within a block command.

```
BLOCK $(...$)
```

The assignments are then expanded in turn. The assignment to `rom` is unchanged. In the assignment to `m`, however, the occurrence of state variable `mpc` is enclosed in a SW expression, to convert it to word kind.

```
m == rom !29 (SW mpc)
```

This is done to all right hand occurrences of state variable names throughout the host program.

The assignments to the locals representing the microword fields are unchanged, though the assignments to the outputs ready and idle are converted to local assignments, as are those to accdisp and pcdisp.

```
ready'   == m[14]
idle'    == m[13]
accdisp' == SW acc
pcdisp'  == SW pc
```

In the assignment to bus, the occurrence of the input switches is converted to a get expression. This is also done with the later occurrences of knob and button. The decimals in the guards of the guarded expression are converted to appropriately sized word constants. By the length rule of the equals operator, their length will be the same as that of the other argument. A default case is also added, assigning a Z value.

```
bus == GUARDED
        || rsw     .= #b1   => GET(switches; time)
        || rpc     .= #b1   => #b000, (SW pc)
        || racc    .= #b1   => (SW acc)
        || rir     .= #b1   => #b000, (SW ir)[12..0]
        || rbuf    .= #b1   => (SW buf)
        || memcntl .= #b01  => (SW mem) !16 (SW mar)
        DEFAULT Z_WORD 16
```

A similar process is performed on the switch expressions in the assignments to each state variable. Here each label should have the same length as the subject of the switch. The assignments are also converted to local assignments. Hence, the assignment to mem becomes

```
mem' ==  SWITCH memcntl
          || #b10 => (SW mar) != (SW bus)
          DEFAULT Z_WORD 131072
```

The assignments to mar and pc are slightly more complex. Both state variables are 13 signals long, but the switch expression assigned has the length of the bus: 16 bits. It is truncated.

```
mar' == (SWITCH...)[12..0]
pc'  == (SWITCH...)[12..0]
```

In the assignment to buf, the possible results of the switch have different lengths. The first has the length of bus. The second contains a decimal. By the length rule of addition the arguments are the same length, so this is converted to a 16 signal constant. As with the third and fourth cases, the length of the result is 17, since plus and minus give results one longer than the arguments. The truncation rule for switch expressions states that the results are truncated to the length of the smallest. The assignment becomes

```
buf' == SWITCH alucntl
        || #b00  => bus
        || #b01  => (#b_0000_0000_0000_0001 + bus)[15..0]
        || #b10  => ((SW arg) + bus)[15..0]
        || #b11  => ((SW arg) - bus)[15..0]
        DEFAULT Z_WORD 16
```

A similar truncation is performed on the switch expression assigned to mpc. In the case labelled by 3, aaddr, which has length 5, is added to knob, which has length 2. The padding rule for addition states that the shorter operand is zero padded.

```
aaddr + (#b_000, GET(knob; time))
```

The same situation also occurs in the final case.

Checks on the well-definedness of each of the 30 locals are then added.

```
CHECK (WD rom)
CHECK (WD m)
CHECK (WD memcntl)
      .
      .
```

Finally, the output assignments and the parallel assignment modelling the clock tick is added.

```
accdisp == PUT(accdisp; time; accdisp')
pcdisp  == PUT(pcdisp; time; pcdisp')
ready   == PUT(ready; time; ready')
idle    == PUT(idle; time; idle')

PAR
 $(
   arg  := MERGE(arg; arg')
   ir   := MERGE(ir; ir')
   buf  := MERGE(buf; buf')
   mar  := MERGE(mar; mar')
   pc   := MERGE(pc; pc')
   acc  := MERGE(acc; acc')
   mem  := MERGE(mem; mem')
   mpc  := MERGE(mpc; mpc')
   time := INC time
 $)
```

7.4 Preprocessing the Expanded Host Program

7.4.1 Expression Simplification

A limited amount of expression simplification can be performed on the expanded host program of Gordon's computer. To allow the control store contents to be in a readable, formated form, the constant assigned to `rom` was split into constants having the size of memory words. These were then concatenated together. One expression simplification that can be performed is to reform the whole constant using constant evaluation.

The assignments to `mar` and `pc` in the source host program, assigned a 16 signal word to these 13 signal variables. In the expanded host program, where the assignments are to `mar'` and `pc'`, the extra signals are truncated using selection.

```
mar' == (SWITCH wmar
            || #b_1 => bus
            DEFAULT Z_WORD 16)[12..0]
```

This selection can be moved inside the switch and then combined with the `Z_WORD 16` operator to give `Z_WORD 13`.

```
mar' == SWITCH wmar
           || #b_1 => bus[12..0]
           DEFAULT Z_WORD 13
```

It is important that this simplification be performed. Otherwise, during the collapsing process, assignments of this form would be collapsed forwards, as the conditional not at the outer level. It is less important here since the value of `wmar` will be known during the collapsing process and so the conditional will itself collapse away.

7.4.2 Check Removal

The second stage of preprocessing involves removing checks of the well-definedness of locals which are guaranteed not to return U values. The commands are scanned in order, starting with the assignment to `rom`. It cannot return a U value since it is assigned a non-U constant. Analysis of the assignments to `m` is slightly more complex. It is assigned the value of a subscription.

```
m == rom !29 (SW mpc)
```

It will be free from U values provided the first argument is free from U values, the last is free from U and Z and the largest possible address is in range, given the size of the first argument. We have already determined that the first argument does not return U values, and since the last is a SW expression it can return neither U nor Z values. `mpc` has length 5 and so can address 32 words of length 29. Since `rom` consists of 32 words of length 29, a U value cannot be returned due to an out of range address. Therefore, `m` cannot return U values.

The control signals take their values from selections on `m`. Since `m` cannot return a U value, neither can any of the control signals. `pcdisp'` and `accdisp'` take values from SW expressions so similarly cannot return U values.

The right hand side of the assignment to `bus` is a guarded expression. It could return a U value if more than one of the guards was active. This depends upon the values held in the control store. In fact, with the given store contents, only one guard will ever be active. However, the simple analysis used cannot ascertain this, and so it is assumed that `bus` could return a U value. Its corresponding check is not removed.

Each of the assignments to the locals associated with the state variables, apart from that to `mpc'`, could assign values from the bus. Since we have not ruled out the possibility of `bus` having a U value, we cannot rule out the possibility of these other locals taking U values.

The assignment to `mpc'` involves switch expressions of the form

```
SWITCH GET(...)...
```

The get expression could return X or Z signals and so `mpc'` could be given a U value, since in these situations the switch would return a U value.

The reduced set of check commands is,

```
CHECK (WD bus)
CHECK (WD mem')
CHECK (WD acc')
CHECK (WD ir')
CHECK (WD arg')
CHECK (WD mar')
CHECK (WD pc')
CHECK (WD buf')
CHECK (WD mpc')
```

7.5 The Skeleton for Gordon's computer

The full state diagram for Gordon's computer is shown in Figure 7.4. The skeleton gives a structured and modular version of this. The macro machine has two modes: an idle and a run mode. A target level machine cycle consists of executing microcode starting and ending at address 0 or 5. Location 0 marks the start of an idle cycle and location 5 of a run cycle.

In the skeleton, a ghost, `Idle`, is introduced to differentiate between these two modes. When `Idle` is `#b1`, an idle cycle will be executed—the microprogram counter, `mpc`, will have value `0`—and when it is `#b0` a run cycle will be executed—`mpc` will have value `5`. This relationship between `Idle` and `mpc` will be specified in the precondition of the skeleton. The jump to the code for the appropriate mode is performed by a case command in the skeleton.

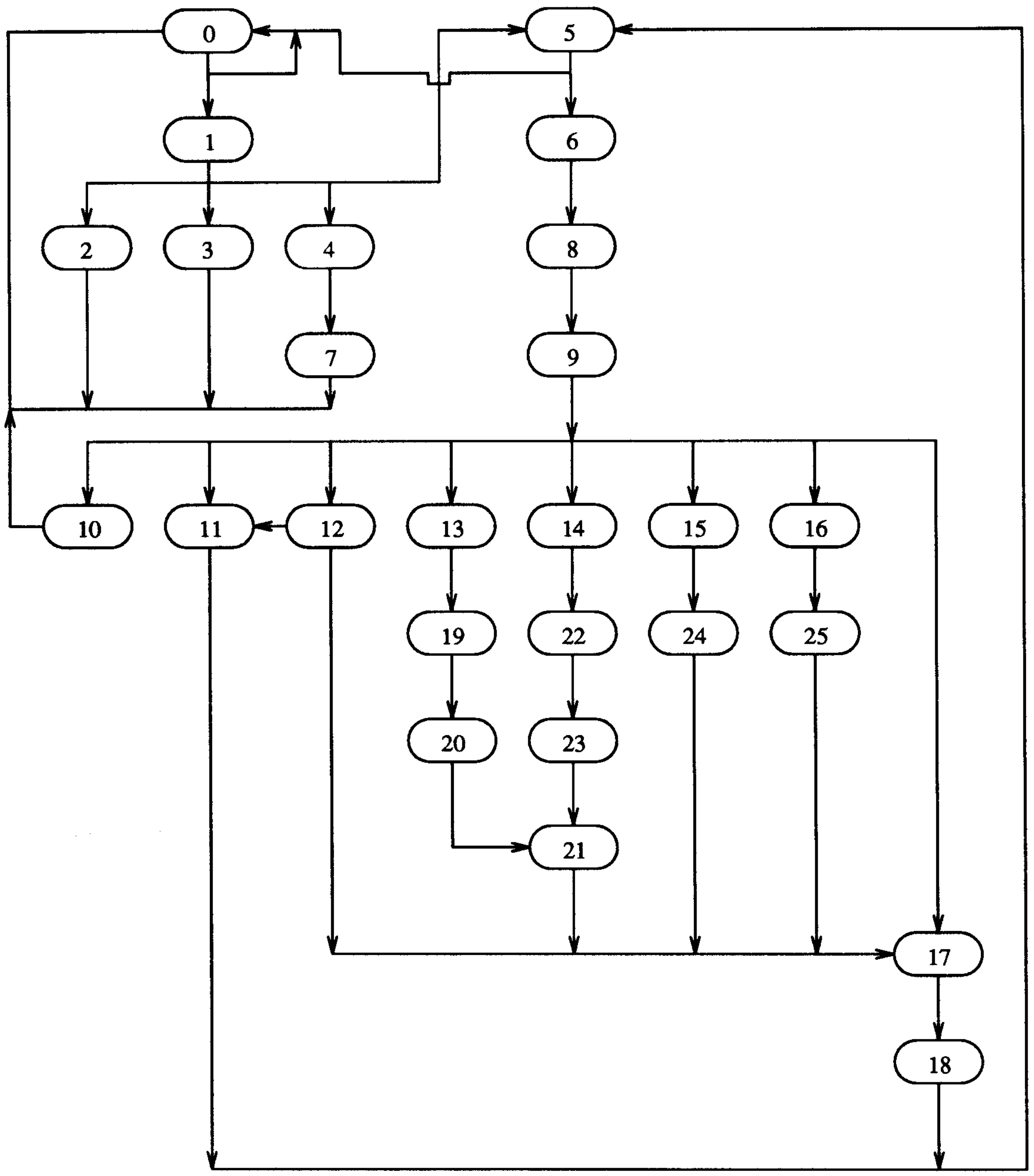

Figure 7.4: The state diagram for Gordon's computer

```
CASE Idle
 || 1 =>  ⟨ idle code ⟩
 DEFAULT  ⟨ run code ⟩
```

7.5.1 Idle Cycles

The first action on an idle cycle will always be the execution of the microinstruction at location 0. This is indicated by an MI command.

```
MI((SW mpc) .= 0) // ready idle jbut 0 1
```

If the value of mpc is not 0 at this point, then the MI command will abort, hence the need for a precondition relating Idle to mpc. The code at location 0 performs a jump, conditional on whether the button is being pressed, either to continue idling or to enter the load phase. This jump is mirrored in the skeleton by a second case command.

```
CASE Button
 || 1 =>  ⟨ load ⟩
 DEFAULT  ⟨ continue idling ⟩
```

Again a ghost is introduced to make the situation clearer. Button has the value of the button at the start of the cycle. The precondition will state this.

If Button is #b0, the machine continues to idle. This is given as a module Idle_to_Idle.

```
DEFAULT CALL_MODULE Idle_to_Idle
```

The computer should just go back to the start of the idle cycle, doing nothing. This is implemented in the skeleton as a SKIP statement. The loading of the value 0 into mpc was performed by the microinstruction at location 0.

```
MODULE Idle_to_Idle
   {...} SKIP {...}
```

The details of the assertions will be considered in a subsequent section.

If the button was pressed at the start of the idle cycle (Button = #b1), then one of the load instructions, or a switch to run mode will be performed. The knob is consulted to determine which function to execute. This switch is made by a microinstruction at location 1. The skeleton models this by a case statement. Each function is given by a separate module.

```
|| 1 =>
   $(
     MI ((SW mpc) .= 1);  // knob 1
     CASE GET (knob; INC Start_Time)
      || 0 => CALL_MODULE Load_pc
      || 1 => CALL_MODULE Load_acc
      || 2 => CALL_MODULE Load_mem
      DEFAULT CALL_MODULE Run
   $)
```

Consider the Run module. It puts the computer into run mode. In the skeleton this involves resetting the ghost Idle. In the microcode, a jump to location 5 is made. This was performed as part of the switch at location 1. The module is

```
MODULE Run
{...}
 Idle == #b0
{...}
```

The Load_mem module executes two microinstructions at locations 4 and 7. Even though the microcode performs a jump, the skeleton code is contiguous. The other modules are similar.

```
MODULE Load_mem
{...}
 $(
   MI ((SW mpc) .= 4); // rpc wmar jmp 7
   MI ((SW mpc) .= 7)  // racc wmem jmp 0
 $)
{...}
```

7.5.2 Run Cycles

Each run cycle starts at microinstruction 5. This is indicated in the skeleton by an MI command.

```
MI ((SW mpc) .= 5) // ready jbut 6 0
```

This tests the button and if pressed at the start of the cycle, switches to idle mode, otherwise it fetches, decodes and executes the next microinstruction.

```
CASE Button
 || 1 => CALL_MODULE Run_to_Idle
 DEFAULT  ⟨ execution stage ⟩
```

The Run_to_Idle module simply changes the value of Idle.

```
MODULE Run_to_Idle
{...}
 Idle == #b0
{...}
```

In the execution stage, the next instruction, as addressed by the program counter, is first fetched. This is done by the code at locations 6 and 8.

```
MI((SW mpc) .= 6); // rpc wmar jmp 8
MI((SW mpc) .= 8); // rmem wir jmp 9
```

The instruction opcode is decoded by microinstruction 9, and a jump made to the code implementing the appropriate macroinstruction. A ghost, Op, represents the value of the next opcode to be decoded. Each macroinstruction is described in a separate module.

```
MI (SW mpc .= 9);         // jop 10
CASE Op
  || 0 => CALL_MODULE Halt
  || 1 => CALL_MODULE Jmp
  || 2 => CALL_MODULE Jze
  || 3 => CALL_MODULE Add
  || 4 => CALL_MODULE Sub
  || 5 => CALL_MODULE Lda
  || 6 => CALL_MODULE Sta
 DEFAULT CALL_MODULE Skp
```

I will consider two modules in detail: Skp and Add. The former macroinstruction has no effect other than to increment the program counter so that the next macroinstruction is fetched. This is performed by code at locations 17 and 18. The latter also performs the jump back to the start of the run cycle.

```
MODULE Skp
{...}
 $(
   MI ((SW mpc) .= 17);  // rpc wbuf inc jmp 18
   MI ((SW mpc) .= 18)   // rbuf wpc jmp 5
 $)
{...}
```

This microcode is shared by many of the macroinstructions, since the incrementation of the program counter is required by any that do not perform jump operations. The code is duplicated in the skeleton so that each macroinstruction has a separate module. This also simplifies the control structure. For example, the Add module is,

```
MODULE Add
{...}
 $(
   MI((SW mpc) .= 13);      // racc warg jmp 19
   MI((SW mpc) .= 19);      // rir wmar jmp 20
   MI((SW mpc) .= 20);      // rmem wbuf add jmp 21
   MI((SW mpc) .= 21);      // rbuf wacc jmp 17
   MI((SW mpc) .= 17);      // rpc wbuf inc jmp 18
   MI((SW mpc) .= 18)       // rbuf wpc jmp 5
 $)
{...}
```

Code at locations 13, 19, 20 and 21 perform the addition, placing the result in the accumulator, then the code at 17 and 18 increments the program counter. This illustrates how the control structure of the microcode has been greatly simplified in the skeleton. The complete body of the skeleton is given in Appendix E.

7.6 Expanding the Skeleton

Expanding the skeleton involves producing collapsed host programs for each of the MI commands. In the skeleton of Gordon's computer, each has the form

MI((SW mpc) .= *w*)

where *w* is some word constant and the substitution used to seed the collapsing process has the form

[WS *w* / mpc]

Consider the Skp module

```
MODULE Skp
{...}
 $(
   MI ((SW mpc) .= 17);  // rpc wbuf inc jmp 18
   MI ((SW mpc) .= 18)   // rbuf wpc jmp 5
 $)
{...}
```

For the first MI command, the substitution is

[WS #b_10001 / mpc]

To perform the collapsing for a particular seed substitution, the preprocessed host program is inspected line by line. The first command is the assignment to rom. This assigns a constant to a local, so a new substitution is formed.

{[#b00_00... / rom], [WS #b_10001 / mpc]}

The need at this stage to convert the original sequence of concatenations into a constant is avoided as it will already have been done by the preprocessing.

The next assignment,

```
m == rom !29 (SW mpc)
```

contains variables with known substitutions. They are replaced by their substitutions yielding a constant expression.

```
(#b_00_...) !29 (SW (WS #b_10001))
```

This may be evaluated to give a constant representing the microinstruction to be executed. The assignment becomes

```
m == #b_00_00_010000_0_01_000_10010_00000_000
```

This command is removed, and the new substitution added to the substitution list.

```
{[#b_00_... / m],...}
```

This substitution is used in each of the assignments to the locals representing the microinstruction fields. Again, the constants are evaluated, the commands removed and the substitutions added.

The assignments to accdisp' and pcdisp',

```
accdisp' == SW acc
pcdisp' == SW pc
```

cannot be simplified further, since no substitution is known for acc or pc. They are just simple assignments, however, so new substitutions are made and the commands removed.

The assignment to **bus** contains the locals **rsw**, **rpc**, **racc**, **rir**, **rbuf** and **memcntl**, the values of which are all known. These values are substituted in for the variables forming constant subexpressions which may be evaluated.

```
bus == GUARDED
          || #b0  .= #b1  => GET(switches; time)
          || #b1  .= #b1  => #b000,(SW pc)
          || #b0  .= #b1  => (SW acc)
          || #b0  .= #b1  => #b000,(SW ir)[12..0]
          || #b0  .= #b1  => (SW buf)
          || #b00 .= #b01 => (SW mem) !16 (SW mar)
          DEFAULT Z_WORD 16
```

All but that originally containing rpc evaluate to #b0. Using the expression rewrite rule, EXP.38(Guarded-zero), they may be removed.

```
bus == GUARDED
          || #b1 => #b000, (SW pc)
          DEFAULT Z_WORD 16
```

This may be simplified further using the rule, EXP.37(Guarded-one), to give the assignment

```
bus == #b000, (SW pc)
```

Once more the command is removed and a new substitution added.

The local associated with state variable `mem`, `mem'`, is assigned the value of a switch expression.

```
mem' ==  SWITCH memcntl
           || #b10 => (SW mar) != (SW bus)
           DEFAULT Z_WORD 131072
```

Substitutions for `memcntl` and `bus` are known. After substitution, the resulting constant subexpressions are evaluated. The value of `memcntl` does not match the label `#b10`, so the rule EXP.34(Switch-default) applies. The command becomes

```
mem' == Z_WORD 131072
```

A similar process is used on each of the other assignments to state variable related locals, allowing the commands to be removed and substitutions formed.

The next group of commands are the checks. For each command, a substitution for the local involved is known and so the local is replaced. Most of the checks were preprocessed away. Of those remaining, for all but `mpc'`, `buf'` and `bus`, the replacement is a Z_WORD expression. The resulting command is, for example, of the form

```
CHECK (WD (Z_WORD 16))
```

The expression rewrite rule, EXP.1(WD), can be used to convert these to

```
CHECK(#b1)
```

since Z_WORD expressions never return U values. The command rewrite rule, COM.1(Check-one), converts these to `SKIP` so they are removed.

The expressions substituted for `bus`, `buf'` and `mpc'` are respectively,

`#b000, (SW pc),`

`(#b_0000_0000_0000_0001 + (#b000, (SW pc)))[15..0]`, and

`#s_10010.`

With simple analysis of the form used when preprocessing, each of these can be seen to be non U so the WD expressions are rewritten to `#b1`. Consequently, all the check commands are removed.

Values are known for each of the output associated locals so these can be substituted for within the output assignments. As the assignments are not local they are not removed. The resulting commands are

```
accdisp == PUT(accdisp; time; SW acc)
pcdisp == PUT(pcdisp; time; SW pc)
ready == PUT(ready; time; #b0)
idle == PUT(idle; time; #b0)
```

Each of the values known for the state variables associated locals can be substituted into the parallel assignment, as can that for mpc. When the substitution is a Z_WORD expression, the merge expression rewrites to the first argument. This leaves assignments within the parallel assignment such as

```
arg := arg
```

Such assignments are removed by the parallel assignment command rewrite rule, COM.2(Par-eq).

The second arguments of the merge expressions assigned to buf and mpc can be seen not to return Z values using the techniques of the preprocessing stage. The right hand sides become

```
WS ((#b_0000_0000_0000_0001 + (#b000, (SW pc)))[15..0])
```

and

```
#s_10010
```

respectively.

The whole preprocessed host program is collapsed to the form

```
BLOCK
 $(
   accdisp == PUT(accdisp; time; SW acc)
   pcdisp  == PUT(pcdisp; time; SW pc)
   idle    == PUT(idle; time; #b0)
   ready   == PUT(ready; time; #b0)
   PAR
    $(
      buf  := WS ((#b_00000000_00000001 +
                    (#b_000, (SW pc)))[15..0])
      mpc  := #s_10010
      time := INC time
    $)
 $)
```

This, together with an appropriate check command, replaces the first MI command in the Skp module. The same process is applied to the other MI command to give the expanded module shown in Figure 7.5.

If it is noted that the MI commands

```
MI ((SW mpc) .= 17)
```

```
MODULE Skp
{...}
 $(
   CHECK((SW mpc) .= #b10001);
   BLOCK
    $(
      accdisp == PUT(accdisp; time; SW acc)
      pcdisp  == PUT(pcdisp; time; SW pc)
      idle    == PUT(idle; time; #b0)
      ready   == PUT(ready; time; #b0)
      PAR
       (
         buf  := WS ((#b_00000000_00000001 + (#b_000, (SW pc)))[15..0])
         mpc  := #s_10010
         time == INC time
       $)
    $);
   CHECK((SW mpc) .= #b10010);
   BLOCK
    $(
      accdisp == PUT(accdisp; time; SW acc)
      pcdisp  == PUT(pcdisp; time; SW pc)
      idle    == PUT(idle; time; #b0)
      ready   == PUT(ready; time; #b0)
      PAR
       $(
         pc   := buf[12..0]
         mpc  := #s_00101
         time := INC time
       $)
    $)
 $)
{...}
```

Figure 7.5: The expanded Skp module

Host Program Version	SIZE	Reduction from the Expanded Host Program
Source	81	42%
Expanded	140	–
Preprocessed	119	15%
Collapsed (largest)	14	90%
Collapsed (smallest)	9	94%
Collapsed (average)	11	92%

Figure 7.6: The SIZE measure for versions of the host program for Gordon's computer

and

```
MI ((SW mpc) .= 18)
```

appear in other modules, the work performed in producing their collapsed host programs need not be repeated.

7.6.1 The SIZE measure

To give quantitative values to the degree of success of the collapsing algorithm, a rough measure, SIZE is used. The SIZE of a host program is the number of lines of pretty printed output of the program. This gives only an approximate idea of the complexity of the program, though it gives a satisfactory picture of the amount of collapsing performed. The measure takes into account both the number of assignments and checks, and also to a limited degree, the complexity of the expressions involved. Complex expressions will generally be formated so as to take up a number of lines. For example, large constants, such as that assigned to `rom` in Gordon's computer are split over multiple lines, as are conditionals which place one guarded clause per line. The measure does not include blank lines, or lines containing just brackets. Also, the declarations are not included.

7.6.2 The SIZE measure for Gordon's computer

The value of the SIZE measure for various versions of Gordon's computer host program are given in Figure 7.6. For the collapsed host programs, the table gives the smallest, largest and average figures for the microinstructions held in the memory of Gordon's computer. They include the check on the condition used to perform the collapsing. From this table it can be seen that the average sized collapsed host program was collapsed by approximately 92% from the expanded host program. Of this 15% was performed in the preprocessing stage due to the removal of check commands. The concise notation allows a reduction of 42% in what the user types from the expanded version of the host program. This difference is due mainly to the checks, extra assignments for state variables and outputs, and extra default cases needed for for many conditional commands.

7.7 The Target Specification

The target specification consists of a precondition, giving conditions intended to hold before execution of the microprogram, and a postcondition giving those intended to hold at the end of the macrocycle. As the skeleton is modular, each macroinstruction is specified separately and given its own precondition and postcondition.

7.7.1 The Main Precondition

As indicated earlier, one of the uses of the precondition is to define the value of ghost variables, relating them to the resources of the machine. This is so with Gordon's computer. Firstly, the time that the macrocycle commences is fixed in a time ghost, `Start_Time`.

```
Start_Time = time
```

The values on the inputs at this time are recorded.

```
Button   = GET(button; Start_Time) /\
Knob     = GET(knob; Start_Time)   /\
Switches = GET(switches; Start_Time)
```

Also the values of the state variables of the macromachine are stored.

```
Mem = SW mem /\
Acc = SW acc /\
Pc  = SW pc
```

Ghost variables are of kind word, and so the state variable values are converted using the SW operator.

The ghost `Op` is used as an abbreviation for the opcode of the instruction pointed to by the program counter at the start of the cycle.

```
Op = (Mem !16 Pc)[15..13]
```

The ghost `Idle` was introduced to differentiate between the two modes of execution: running and idling. In the machine, the mode is determined by the value of the microprogram counter, `mpc` so the value of `Idle` must be related to the value of `mpc`.

```
(Idle = #b0 /\ mpc = #s00101) \/
(Idle = #b1 /\ mpc = #s00000)
```

This also specifies that the only possible values of `mpc` at the start of the cycle are 0 and 5. Since `Idle` is changed by the skeleton, its value at the start of the cycle is stored in a second ghost, `Idle_at_Start`.

```
Idle_at_Start = Idle
```

Restrictions must be placed on the values of inputs and the initial values of state variables. For the host program to operate correctly, the registers pc and acc and the memory, mem, must not contain X values. Since X values do not occur in reality, it is not a real world problem. However, X values do exist in the model, and if they occurred in one of the state variables, then the host program could fail. For example, if pc held an X signal, this could be transferred to the machine address register, mar and a memory read, (mem !32 mar) would fail due to the semantics of subscription. None of the other registers could cause problems as they are always set before use in the microprogram. The predicate BINARYP asserts that its word argument consists of only 0 and 1 signals.

```
BINARYP (SW mem) /\
BINARYP (SW pc) /\
BINARYP (SW acc)
```

This suggests that the initial values of these registers is important and should not be random.

A similar problem occurs for the inputs. At no time (i.e., during no microcycle) should they hold signals other than 0 or 1. The predicate INPUT_BINARYP asserts this of its history word argument. This does correspond to a real restriction as it also states that the inputs are never given high impedance (Z) values by the outside environment.

```
INPUT_BINARYP (button) /\
INPUT_BINARYP (knob) /\
INPUT_BINARYP (switches)
```

This precondition gives a restriction on the values input for *all* time, not just the present.

7.7.2 The Main Postcondition

The main postcondition for Gordon's computer consists of a disjunction of predicates, one for each macroinstruction. The use of predicates in this way modularises the specification. By the end of the cycle, one of the macroinstructions should have been executed. We would also like to be certain that the correct one was executed. Each predicate should therefore include the path condition that must hold for it to be the correct one. This is given in terms of the ghost variables which record the values from the start of the cycle.

```
|- CYCLE_END (...) =
       ((IDLE_TO_IDLE (...) \/
        (RUN_TO_IDLE (...) \/
                .
                .
```

These predicates are also used in the postconditions for each macroinstruction module. Consider the IDLE_TO_IDLE macroinstruction in which the computer completely idles. The specification with respect to the target resources states that they are unaltered. The following condition will be included in the body of the predicate IDLE_TO_IDLE:

```
SW mem = Mem /\
SW pc  = Pc  /\
SW acc = Acc
```

The computer should also remain in the idle mode.

```
Idle = #b1
```

We only wish this predicate to be true when an IDLE_TO_IDLE macroinstruction should have just been executed, that is when the computer was in idle mode and the button was not pressed.

```
Idle_at_Start = #b1 /\
Button        = #b0
```

As an example of a specification for a run macroinstruction, consider the SKP macroinstruction. This requires that the program counter is incremented, the accumulator and memory are left unchanged, and the computer remains in the run mode.

```
SW mem = Mem /\
SW pc  = (Pc + #b_0000_0000_0000_1)[12..0] /\
SW acc = Acc /\
Idle   = #b0
```

A SKP instruction should be executed if the computer is in the run mode, the button is not pressed and the opcode of the instruction fetched is 7.

```
Idle_at_Start = #b0 /\
Button        = #b0 /\
Op            = #b111
```

The specifications for the load instructions are slightly more complex. They read the knob and/or switches at some unspecified point during the macrocycle, and so these inputs should not be changed during this period. At the target level, we are working at the target level time scale. It would therefore be inappropriate to subdivide the time at which these signals should be stable below this scale. This condition is given as a predicate KNOB_AND_SWITCHES_STABLE defined in terms of a second predicate STABLE.

```
|- STABLE (t1; t2; h) = ∀t.  t1 < t < t2 ==>
                              GET (h; t) = GET (h; t1)
```

used instead.

```
|- KNOB_AND_SWITCHES_STABLE (t1; t2; s; k) =
            STABLE (t1; t2; s) /\
            STABLE (t1; t2; k) /\
```

The specification for LOAD_PC is

```
Idle_at_Start = #b1 /\
Button        = #b1 /\
(KNOB_AND_SWITCHES_STABLE (Start_Time; time; switches; knob) ==>
               Knob   = #b00 /\
               SW mem = Mem /\
               SW pc  = Switches[12..0] /\
               Idle   = #b1
```

If the knob and switches are changed during the instruction, the effect is left unspecified.

The above specifications describe the macrolevel view. We would also like to verify some facts about the microlevel. Since the microcode sits in a continuous loop executing microcycles, we would like to be sure that at the end of the cycle, the computer is in a suitable state to execute the next cycle. This condition corresponds to that part of the precondition which does more than just define ghost variables, notably, that the value of mpc and Idle are linked; that mpc has value 0 or 5, and that the state variables have only binary values. It is the loop invariant for the infinite loop that the microprogram executes. It is given by the predicate LOOP_INVARIANT. The input history values cannot be affected in any way by the program, due to the enforced syntactic structure of the host program itself. They are, therefore, not included in the postcondition.

Certain conditions must also hold about the idle and ready lights. The ready light should only be on at the start of each macrocycle.

```
GET (ready; Start_Time) = #b1 /\
LOW (Start_Time; time; ready)
```

Here the predicate LOW states that the history value of its third argument has value #b0 between the times given by its first two arguments.

```
|- LOW (t1; t2; h) = ∀t.  t1 < t < t2 ==> GET (h; t) = #b0
```

As with the definition for STABLE, this definition cannot be input to the system. The idle light should be on at the start of an idle cycle and off at all other times.

```
Idle_at_Start = GET(idle; Start_Time) /\
LOW (Start_Time; time; idle)
```

These conditions are given by predicates READY_CONDITION and IDLE_CONDITION respectively.

The above requirements use the microlevel time scale. The lights are intended to be on for single microcycles, only. This is because they were partly included in the design to help the verification in the earlier proofs. The ready signal is used to mark the boundaries of the macrocycles, acting as a target level time scale clock. For the SPOOK verification the structure of the skeleton itself indicates these boundaries. Similarly, the ghost Idle plays the role of the idle light, indicating the mode. This illustrates an occasion when the SPOOK system does not require the design to be affected by the need for verification.

The postcondition is extended to include the extra conditions.

```
|- CYCLE_END (...) =
                  .
                  .
         /\ IDLE_CONDITION (...)
         /\ READY_CONDITION (...)
         /\ LOOP_INVARIANT (...)
```

7.7.3 The Module Specifications

Each module has its own precondition and postcondition, allowing it to be independently verified. In Gordon's computer, the modules correspond to the various macroinstructions. The postcondition is given by the appropriate disjunct used in the main postcondition, together with the conditions on the lights and the loop invariant. For the Skp module it is given by

```
{ SKP (...) /\
IDLE_CONDITION (...) /\
READY_CONDITION (...) /\
LOOP_INVARIANT (...) }
```

Each of the module postconditions have a similar form.

The preconditions describe the initial environment, from which the module should be executed. They must contain enough information for the corresponding postcondition to be proved and they must contain information about any values set up in the micro-level registers, such as ir, before their call. They give documentation about the implementation, rather than just giving a specification of that intended to be implemented.

Consider the precondition for the IDLE_TO_IDLE macroinstruction. It must assert that the appropriate path condition for the macroinstruction to be executed is true.

```
Idle_at_Start = #b1 /\
Button        = #b0
```

Information relating the values of the target resources to the initial values must be asserted.

```
SW mem = Mem /\
SW pc  = Pc /\
SW acc = Acc /\
Idle   = #b1
```

The microprogram counter should have value 0 when the module is called—the address of the next microinstruction to be executed.

```
mpc = #s_00000
```

A more general condition

```
mpc == WS(SWITCH Button || #b1 => #b00001 DEFAULT #b00000)
```

is actually used to allow the same predicate to be used in several places.

Information sufficient to prove the loop invariant is also required. This means that the conditions stating that the state variables have not been given X values must be included.

Finally, the postcondition requires that for the whole of the cycle, the conditions on the ready and idle lights have been maintained. The precondition asserts that this has been so during the cycle up to the point when the module was called. This also requires information about the current time relative to the time at the start of the cycle.

```
READY_CONDITION (Start_Time; time; ready) /\
IDLE_CONDITION (Idle_at_Start; Start_Time; time; idle) /\
time = INC Start_Time
```

The run macroinstructions have similar preconditions. The path condition requires information about the value of the opcode in addition to the initial mode and whether the button was pressed. For the Skp module the path condition also includes the following condition.

```
Op = #b111
```

The instruction register is intended to hold the current instruction.

```
ir = WS (Mem !16 Pc)
```

Also, the microprogram counter must have as value the next microinstruction to be executed—17 for the SKP instruction.

```
mpc = #s01101
```

Again, a more general condition is actually used in this case, giving the value of the microprogram counter in terms of the instruction register's.

```
mpc = WS((#b01010 + (#b00, (SW ir)[15..13]))[4..0]
```

This allows the same predicate, RUN_PRE_CALL, to be used in the preconditions of each of the run modules and in the annotation required before the case command that calls them.

7.7.4 Annotations

Annotations are placed in sequences before conditional, loop and module calling commands. They serve to further document the skeleton as well as being required to aid the generation of verification conditions. It turns out that just five annotations are required for Gordon's computer; one before each of the sequenced case commands in the main body of the skeleton and one before the case command in the Jze module. These annotations describe the state at the particular point in the skeleton and must carry enough information to prove the later annotations. Each is defined as a predicate. This helps keep the skeleton readable since the annotations are fairly voluminous. Also the same predicates can be used in the appropriate module preconditions. These have the additional path condition corresponding to the case command. For example, the predicate RUN_PRE_CALL is used as the annotation prior to the condition in the case command which switches between the possible run macro-commands, as mentioned in the previous section. The precondition for the Skp module then has the form below.

```
RUN_PRE_CALL (...) /\
Op = #b111
```

7.8 Producing Verification Conditions

The verification conditions for a module are produced by applying the VCGEN tactic based on the verification condition generation rules. Consider once more the expanded Skp module of Figure 7.5. It consists of a sequence at the top level of the form,

$\{P\}\ C_1$; BLOCK $C_2\ \{Q\}$

where Q is the condition,

```
SKP (Idle_at_Start; Button; Op; mem; Mem;
     pc; Pc; acc; Acc; Idle) /\
IDLE_CONDITION (Idle_at_Start; Start_Time; time; idle) /\
READY_CONDITION (Start_Time; time; ready) /\
LOOP_INVARIANT (Idle; mpc; mem; pc; acc; button; knob; switches)
```

Since no annotation is provided after the semicolon, the second sequencing rule, VCGEN.3, applies. The value of

$$\mathsf{WP}(\texttt{BLOCK}\ C_2,\ Q)$$

must be determined. By the definition of WP, this is

$$\mathsf{WP}(C_2,\ Q[U/locals])[U/locals]$$

which is equal to

$$\mathsf{WP}(C_2,\ Q)[U/locals]$$

since Q does not contain any locals.

Now C_2 has the form

```
C3;
PAR
 $(
    pc := buf[12..0]
    mpc := #s_00101
    time == INC time
 $)
```

so by the definition of WP for sequencing and parallel assignment the following equality holds.

$$\begin{aligned}\mathsf{WP}(C_2,\ Q) &= \mathsf{WP}(C_3,\ \mathsf{WP}(\texttt{PAR}\ldots,\ Q))\\ &= \mathsf{WP}(C_3,\ Q[\texttt{INC time/time},\ \texttt{\#s\_00101/mpc},\ \texttt{buf[12..0]/pc}])\end{aligned}$$

The definition for WP is continued to be expanded in this way, until ultimately a value for WP(BLOCK C_2; Q) is obtained:

```
Q[INC time/time , #s_00101/mpc , buf[12..0]/pc]
 [PUT(ready; time; #b0)/ready]
 [PUT(idle; time; #b0)/idle]
 [PUT(pcdisp; time; SW pc)/pcdisp]
 [PUT(accdisp; time; SW acc)/accdisp]
 [U/locals]
```

Making the substitutions to Q yields the assertion,

```
SKP
  (Idle_at_Start; Button; Op; mem; Mem;
   buf[12..0]; Pc; acc; Acc; Idle) /\
IDLE_CONDITION
  (Idle_at_Start; Start_Time; INC time; PUT(idle;time;#b0)) /\
READY_CONDITION
  (Start_Time; INC time; PUT(ready;time;#b0)) /\
LOOP_INVARIANT
  (Idle; #s_00101; mem; buf[12..0]; acc; button; knob; switches)
```

From the sequencing rule the verification conditions for the **Skp** module are those for

$\{P\}\ C_1\ \{\text{WP}(\texttt{BLOCK}(C_2;Q))\}$

C_1 is itself a sequence

C; CHECK (...)

so the process can be repeated once more. Ultimately the following verification condition is generated.

```
(RUN_PRE_CALL
   (Idle; Button; Mem; Pc; Idle_at_Start; Start_Time; Op;
    mpc; ir; mar; mem; pc; acc; time;
    button; knob; switches; idle; ready) /\
Op = #b111) ==>

SKP
   (Idle_at_Start; Button; Op; mem; Mem;
    WS ((#b_00000000_00000001 +
          (#b_000, (SW pc)))[15..0])[12..0];
    Pc; acc; Acc; Idle) /\
IDLE_CONDITION
   (Idle_at_Start; Start_Time; INC (INC time);
    PUT(PUT(idle;time;#b0);(INC time);#b0)) /\
READY_CONDITION
   (Start_Time; INC (INC time);
    PUT(PUT(ready;time;#b0);(INC time);#b0)) /\
LOOP_INVARIANT
   (Idle; #s_00101; mem;
    WS ((#b_00000000_00000001 +
          (#b_000, (SW pc)))[15..0])[12..0];
    acc; button; knob; switches) /\
IS_ONEP(SW #s10010 .= #b10010) /\
IS_ONEP(SW mpc .= #b10001)
```

The first four conjuncts of the consequent, which consist of the predicates, SKP, IDLE_CONDITION, READY_CONDITION and LOOP_INVARIANT, respectively, arise directly from the occurrences of these predicates in the postcondition. The latter two conjuncts are produced for the two check commands. If they can be shown true under the assumption of the antecedent, then the control structure of the module is correct.

7.9 Proving Verification Conditions

Much of the work required to prove the verification conditions can be performed within the system using the AUTO_THM tactic. Consider the verification condition generated for the SKP module. Initially the antecedent is split into separate assumptions, and separate subgoals are formed for each conjunct in the consequent. This gives six subgoals

```
A-1 { RUN_PRE_CALL
        (Idle; Button; Mem; Pc; Idle_at_Start; Start_Time; Op;
         mpc; ir; mar; mem; pc; acc; time;
         button; knob; switches; idle; ready),
    Op = #b111 } |-
    SKP
       (Idle_at_Start; Button; Op; mem; Mem;
        WS ((#b_00000000_00000001 + (#b_000, (SW pc)))[15..0])[12..0];
        Pc; acc; Acc; Idle)

A-2 {...} |-
    IDLE_CONDITION
      (Idle_at_Start; Start_Time; INC (INC time);
       PUT(PUT(idle;time;#b0);(INC time);#b0))


A-3 {...} |-
    READY_CONDITION
      (Start_Time; INC (INC time);
       PUT(PUT(ready;time;#b0);(INC time);#b0))


A-4 {...} |-
    LOOP_INVARIANT
      (Idle; #s_00101; mem;
       WS ((#b_00000000_00000001 + (#b_000, (SW pc)))[15..0])[12..0];
       acc; button; knob; switches)


A-5 {...} |- IS_ONEP(SW #s10010 .= #b10010)


A-6 {...} |- IS_ONEP(SW mpc .= #b10001)
```

The axioms do not match any of these goals, and the only substitution using the tactic EQSUB that can be made is that for the value of Op in the first (A-1). The fifth (A-5) contains a constant expression, which can be evaluated to #b1. The resulting assertion can, in turn, be rewritten with the condition rewrite rule, COND.3(Is_onep_T), giving

```
{...} |- T
```

This matches the simple rule ALL_ENTAILS_T and so generates no subgoals.

For each of the other subgoals nothing further can be done without opening up the predicate definitions. Only those predicates suggested by the user are expanded. This means that if no predicates were originally named to the AUTO_THM tactic, this would be all that was initially done. This would allow the user to then look at each subgoal separately and if errors occurred the user would have a better feel for their cause. Alternatively, the predicates could be named at the start, allowing the proof to continue without user assistance at this point.

Consider Subgoal (A–1). Here the predicates SKP and RUN_PRE_CALL can be expanded. Figure 7.7 gives the resulting goal after the individual conjuncts have been divided into separate assumptions. This splits into seven subgoals.

```
B–1 {...} |- Idle_at_Start = #b0

B–2 {...} |- Button = #b0

B–3 {...} |- Op = #b111

B–4 {...} |- SW mem = Mem

B–5 {...} |-
          SW (WS ((#b_00000000_00000001 +
                  (#b_000, (SW pc)))[15..0])[12..0]) =
          (Pc + #b0000000000001)[12..0]

B–6 {...} |- SW acc = Acc

B–7 {...} |- Idle = #b0
```

Only the fifth, B–5, is not easily proved. For the others an equality substitution can be performed, and then, after rewriting, one of the axioms applies. For example, in Subgoal, B–6,

```
    {..., Acc = SW acc,...} |- SW acc = Acc
```

the expression (SW acc) is substituted for Acc yielding

```
    {...} |- SW acc = SW acc
```

The condition rewrite rule, COND.13(E_eq_E), applies, rewriting this to

```
    {...} |- T
```

which matches the simple rule ALL_ENTAILS_T.

Subgoal (B–5) can be converted to the goal

```
    {...}|-
        (#b_00000000_00000001 + (#b_000, (SW pc)))[12..0] =
        ((SW pc) + #b00000_00000001)[12..0]
```

However, the rules are insufficient to simplify this further. This goal must be proved outside the system. It can be fairly simply seen to be true. First the commutativity of the plus operator is used.

```
{
Idle           = #b0 ,
Button         = #b0 ,

Mem            = SW mem ,
Acc            = SW acc ,
Pc             = SW pc ,

Idle_at_Start = #b0 ,
Op             = (Mem !16 Pc) [15..13] ,

mpc            = WS((#b01010 + (#b00, (SW ir)[15..13]))[4..0]) ,
ir             = WS(Mem !16 Pc) ,
mar            = WS Pc ,
time           = INC(INC(INC(INC Start_Time))) ,

BINARYP(SW mem) ,
BINARYP(SW pc) ,
BINARYP(SW acc) ,

INPUT_BINARYP (button) ,
INPUT_BINARYP (knob) ,
INPUT_BINARYP (switches) ,

IDLE_CONDITION(Idle_at_Start; Start_Time; time; idle) ,
READY_CONDITION(Start_Time; time; ready),
Op             = #b111 }
|-
  Idle_at_Start = #b0 /\
  Button         = #b0 /\
  Op             = #b111 /\
  SW mem         = Mem /\
  SW (WS ((#b_00000000_00000001 + (#b_000, (SW pc)))[15..0])[12..0]) =
                 (Pc + #b0000000000001)[12..0] /\
  SW acc         = Acc /\
  Idle           = #b0
```

Figure 7.7: Subgoal A–1 from the SKP module after predicate expansion

```
{...}|-
     (#b_00000000_00000001 + (#b_000, (SW pc)))[12..0] =
     (#b00000_00000001 + (SW pc))[12..0]
```

Then, the selection operator is moved inside the addition on the left hand side of the equality.

```
{...}|-
     (#b_00000000_00000001[12..0] +
      (#b_000, (SW pc))[12..0])[12..0] =
     (#b00000_00000001 + (SW pc))[12..0]
```

The rules already in the system would then be able to prove this goal. Thus, with the addition of two fairly simple rewrite rules, this goal would be automatically provable by the system.

Returning to the original subgoals, those involving `IDLE_CONDITION` (A–2) and `READY_CONDITION` (A–3) can be proved more easily by proving general theorems about the predicates since many similar goals arise when proving the verification conditions of other modules. Consequently, these predicates are not expanded and the goals are left to be proved externally.

The predicate `LOOP_INVARIANT` in Subgoal (A–4) is expanded in the same way as `SKP` yielding a large number of easily proved subgoals. All but one of these can be proved.

The final Subgoal (A–6), is not proved because the heuristic used does not contain enough intelligence to make suitable substitutions from the assumptions. Again, it can be quite easily be seen to be true outside the system.

Five unproved subgoals remain. These are significantly simpler than the originals. This illustrates how the majority of the subgoals that require proving are of a very simple nature. A large amount of the theorem proving work can easily be done automatically.

7.9.1 Statistics for Measuring the Success of the Theorem Prover

Statistics are provided to indicate the success of the automatic theorem prover when proving verification conditions. The basis of these statistics is a measure GOAL_COUNT which gives an indication of the number of subgoals which must be proved to prove a verification condition. For the purposes of the statistics, the number of subgoals associated with a verification condition is considered to be the total number of conjuncts at the top level in the consequent including those in the definition of any predicates used.

GOAL_COUNT $[\![\theta$ |- $A \supset B]\!]$ = GOAL_COUNT $[\![\theta \cup A$ |- $B]\!]$
GOAL_COUNT $[\![\theta$ |- A /\ $B]\!]$ =
GOAL_COUNT $[\![\theta$ |- $A]\!]$ + GOAL_COUNT $[\![\theta$ |- $B]\!]$
GOAL_COUNT $[\![\theta$ |- $A]\!]$ = 1
where A is not a predicate or a conjunction.
GOAL_COUNT $[\![\theta$ |- $P]\!]$ = GOAL_COUNT $[\![\theta$ |- body of $P]\!]$
where P is a predicate.

This measure gives a fairly good indication of how many subgoals must be proved to prove a verification condition, though gives no indication of the effort required to prove these subgoals or generate them.

Simply giving the number of verification conditions generated and proved by the system would not give a very good picture since in general the system proves most, but not all of the conjuncts in the verification condition, leaving the remainder to be proved outside the system. It is quite likely that such a measure would indicate that no work had been performed.

The GOAL_COUNT measure does give an inaccurate picture for certain goals. For example, a goal of the form

P |- P

where P is a predicate, can be proved immediately from the axiom A_ENTAILS_A. The measure GOAL_COUNT, however, indicates that the number of subgoals which must be proved depends on the number of conjuncts in the definition of P. This is not completely inaccurate, since on the whole predicates are used in the style of macros to increase the readability of the assertions. The way the above goal is proved could be thought of as just a trick to prove many subgoals together.

The GOAL_COUNT measure only gives an indication of the success in proving the verification conditions that are actually generated. One of the main features of the SPOOK system is that much simplification is performed at an earlier stage in the proof process. Work that would otherwise by done at the theorem proving stage is performed before the verification conditions are generated and so does not appear in the GOAL_COUNT statistics. For example, a verification condition will be generated for each check on the well-definedness of a local in the host program for each MI command. Since the majority of these checks are removed at the collapsing stage, prior to the generation of verification conditions, it will not appear in the statistics.

The GOAL_COUNT Measure for the Verification of Gordon's computer

The GOAL_COUNT statistics for Gordon's computer, are given in Figure 7.8.

In addition to giving statistics for each module and the main body of the skeleton separately, the totals for the whole proof are given. The first column of statistics indicates the number of subgoals which must be proved to prove the particular module and the second gives the number remaining after the

Module	Initial GOAL_COUNT	Remaining GOAL_COUNT	Percent Proved
Idle_to_Idle	12	1	92
Run_to_Idle	12	1	92
Load_Pc	14	4	64
Load_Acc	14	5	64
Load_Mem	16	5	69
Run	13	1	92
Halt	14	3	79
Jmp	14	4	71
Jze	38	14	63
Add	19	5	74
Sub	19	6	68
Lda	17	6	65
Sta	18	7	61
Skp	15	5	67
Main	459	26	94
Total	694	94	86

Figure 7.8: The GOAL_COUNT statistics for the verification of Gordon's computer

automatic tactic **AUTO_THM** has been used. The final column gives the percentage proven automatically. The statistics take into account the definitions of all predicates defined, apart from **KNOB_AND_SWITCHES_STABLE**, **READY_CONDITION** and **IDLE_CONDITION**. These were omitted since they were not provided to the **AUTO_THM** tactic.

The figures show that even with a fairly naive theorem prover, the vast majority of the goals (86%) are of a simple enough nature to be proved automatically. The remainder were also of a simple nature. Some could not be proved because they involved predicates such as **READY_CONDITION**, for which the system had no information. Others could not be proved because of the simplistic heuristic used when extracting equality information from the assumptions. A further group were not proved due to the rewrite rules provided being insufficient. For many unproven goals, simple additions to the system would have been sufficient to allow them to be proved. Furthermore, the necessary additions would appear to be useful for several of the unproved goals. They would also appear to be general enough to be of use in the proofs of microprograms for other architectures. It is clear that with a more powerful, state of the art theorem prover, given sufficient information about the SPOOK operators and predicates defined, a much greater percentage of the theorem proving work could have been performed automatically.

7.10 Errors found

Several errors were found in the various specifications during the initial verification of Gordon's computer, including one in the actual microprogram. Each is detailed below. These errors were corrected and the proof reperformed. The location and reason for each error was fairly easy to locate due to the modular nature of the proof.

7.10.1 The Microprogram

An error was discovered in the version of the microprogram being verified. It occurs in the version given in the BSPL report [Richards 86] and is not present in the microprograms verified using other methods. The `aaddr` field of the instruction at ROM location 1 is incorrect. It should contain the constant `#b_00010` corresponding to a `knob 1` assembler instruction rather than `#b_00001` (`knob 0`). The assembler version in the report is actually correct, and the error was probably made due to the unobvious assembler notation. A `knob 1` instruction actually means the base address is 2. This highlights the problems of verifying an assembler version of the program rather than the actual memory image. Assembly level verification systems would have missed the error since the assembly program is correct. The mistake was introduced during the assembly process, which was done by hand. In an initial rough hand proof performed while developing the SPOOK system, this error was missed. This was because the collapsed host programs were translated directly from the assembler program, in a fashion similar to that used by the assembly level systems.

The error was found due to a verification condition being unprovable. The microprogram error was in fact mirrored in the preconditions of the modules implementing the idle operations, such as loading the program counter from the switches. For this reason, the unprovable verification condition was generated from a check command. The precondition stated that `mpc` had one value, whereas a different value had been used to perform the collapsing of the first instruction on that path. The `Load_pc` module had the form

```
{ mpc = #s_00001 /\ ...}
MI(SW mpc .= #b_00010)
{...}
```

This expands to

```
{...}
CHECK((SW mpc) = #b_00010);
BLOCK $(...$)
{...}
```

The verification condition generated from the check command had the form

```
{...} |- #b_00001 = #b_00010
```

which could not be proved. This illustrates the way in which the control structure specified by the skeleton is checked with that which actually occurs. Had the preconditions been correct, the error would have been caught due to a verification condition generated from the `CALL_MODULE` command being unprovable.

Because of the simple nature of Gordon's computer, and the fairly vital point in the microprogram that it occurred, this mistake would probably have been discovered by simulation much more simply, had any been performed. For a more complex architecture, more obscure errors could easily be missed by simulation. Verification should be performed in conjunction with simulation to enable errors to be found most economically. Since the SPOOK language is executable, the system could be used in this way.

7.10.2 The Host Program

The host program was converted from the BSPL specification of Gordon's computer. Several errors were discovered in the BSPL specification, though by visual inspection and type checking rather than by unprovable verification conditions.

1. In the BSPL specification the variable `stopped` is initialised but is not declared or subsequently used. It does not correspond to anything in the micromachine. This was noted during initial inspection of the specification, when the SPOOK specification was being prepared. Currently SPOOK does not provide an equivalent to BSPL's `INITIAL_VALUES`, so this error does not affect the SPOOK specification.

2. The microprogram given in the BSPL specification must be extended to addresses 26 to 31 with `jmp 0` instructions so that the machine will eventually start operating correctly, whatever state the microprogram counter powers up in. This error was present in the original description of Gordon's computer [Gordon 81b], but was corrected in the later proof conducted using HOL [Joyce 86]. No SPOOK proof has been performed to show that the machine will ultimately arrive in a suitable initial state after power up. This could be done by providing a module giving the appropriate paths for each possible initial value of `mpc`.

3. Expressions `(SW pc)` and `((SW ir)[12..0]` have length 13 but are assigned to the bus which has length 16. In both SPOOK and BSPL it is an error for the value assigned to be of shorter length than the variable assigned to. This was detected during type checking. It was corrected by zero padding the expressions, i.e., using `(#b000, (SW pc))` and `(#b000, (SW ir)[12..0])`.

7.10.3 Target Specification

The original postcondition of the `Jze` module for the path when the accumulator is zero, included an incorrect conjunct:

```
Idle = #b1
```

This conjunct specified that after executing a JZE instruction with a zeroed accumulator, the machine should move into an idle state. This contradicts the informal specification, and in fact the implementation. The conjunct should have been

```
Idle = #b0
```

since the machine should continue to execute the program. This was detected due to an unprovable verification condition of the form

```
{...} |- ~T
```

being generated.

This illustrates how easily specifications can contain errors. Had the microcode been written to implement the formal specification, the proof would have gone through despite the microprogram being incorrect. The proof only shows that the implementation and specification are consistent. For this reason it is possibly better to ultimately think of the specification being documentation of what the code does, rather than as a specification of what it is intended to do.

It also suggests that it would be useful to be able to perform simulations on the target specification in addition to the host and microprogram. This cannot be done in the SPOOK system since the predicate calculus is used as the specification language. The problem could possibly be overcome by restricting the specification to executable subsets of the predicate calculus. Alternatively an executable language could be used from which the axiomatic form of the target could be generated. Axiomatic definition languages such as AADL[Damm 88] could be used in this way. This would have the additional advantage that the target specification would be more readable.

7.10.4 The Skeleton

The skeleton module for the JZE instruction was originally incorrect. The two branches in the case command were originally reversed. The skeleton suggested that the jump was made when the accumulator was non-zero, rather than zero as specified and implemented. It should have been

```
CASE Acc
  || 0 => // Move to next instruction
       MI((SW mpc) .= 11) // Perform jump
  DEFAULT
    $(
      MI((SW mpc) .= 17);
      MI((SW mpc) .= 18)
    $)
```

but was actually

```
CASE Acc
  || 0 => // Move to next instruction
       $(
         MI((SW mpc) .= 17);
         MI((SW mpc) .= 18)
       $)
  DEFAULT MI((SW mpc) .= 11) // Perform jump
```

This was discovered because a verification condition of the form

```
{...} |- (Mem !16 Pc)[12..0] = (Pc + 1)[12..0]
```

was obtained, but could not be proved. The left hand side of the equality suggests that a jump is made, whereas the right just suggests that execution moves to the next instruction. In this way, the unprovable verification condition gave clues as to the location of the error. It was already known to be in the JZE instruction, as that was the module being verified. This illustrates the way that the skeleton, and consequently the users view of the control structure, is checked as part of the verification process. If the state transition information had been given informally, and not checked in this way, the error in the documentation might not have been discovered.

7.10.5 The Annotations

During the development of the proof many mistakes were discovered in the annotations. On the whole, these tended to be due to too little information being included to prove facts in subsequent annotations. Often this was due to there being unnecessary information in the later ones.

For example, the clause

```
Button = #b0
```

was placed in the annotation JZE_PRE_BRANCH, which occurs before the jump is made, but was not in RUN_PRE_CALL, the annotation occurring before the instruction switch was made. It could, therefore, not be proved in the verification condition generated for the later annotation. In fact, it was not needed to prove the postcondition for the Jze module and so was not needed in the annotation.

The placing of annotations is very much an iterative task. The modular nature of the proof aided this, as changing annotations in one module did not invalidate proofs of other modules. It also suggests that it is not wise to use information from the annotations to perform the collapsing of the host or expanded skeleton, since each time the annotation was changed, the skeleton would have to be re-expanded.

Chapter 8

The Orion Computer

In this chapter, a case study of the HLH Orion computer is described. This is a commercially produced machine with a fairly complex microarchitecture. This example illustrates that the techniques scale to production microarchitectures. The host program for the Orion and the ramifications of using a skeleton with respect to this microarchitecture are considered. The standard instruction set of the computer is then discussed and in particular, the verification of the And instruction described. Statistics illustrating the success of the system are given.

8.1 Introduction

In this chapter I consider the application of the techniques described in this dissertation to a fragment of production microcode for a production computer. The computer chosen was the High Level Hardware Orion Computer. This is a microprogrammed 32 bit computer, based on the Am2901 bit slice chip set using a 2910 microprogram controller. The block diagram of its Central Processing Unit (CPU) is shown in Figure 8.1. The Orion control word is 64 bits wide and has a diagonal format, consisting of 18 fields and a parity bit. On the whole, the format is horizontal, except for some infrequently used and mutually exclusive functions which are encoded within single fields vertically. A pipeline is used to fetch microinstructions from the control store. The Orion supports virtual memory and has a sizable cache.

The High Level Hardware Orion Computer was selected because

- it provides a realistic example,
- a formal specification of the microarchitecture was available, and
- assembler versions of the microcode used in the standard system were available.

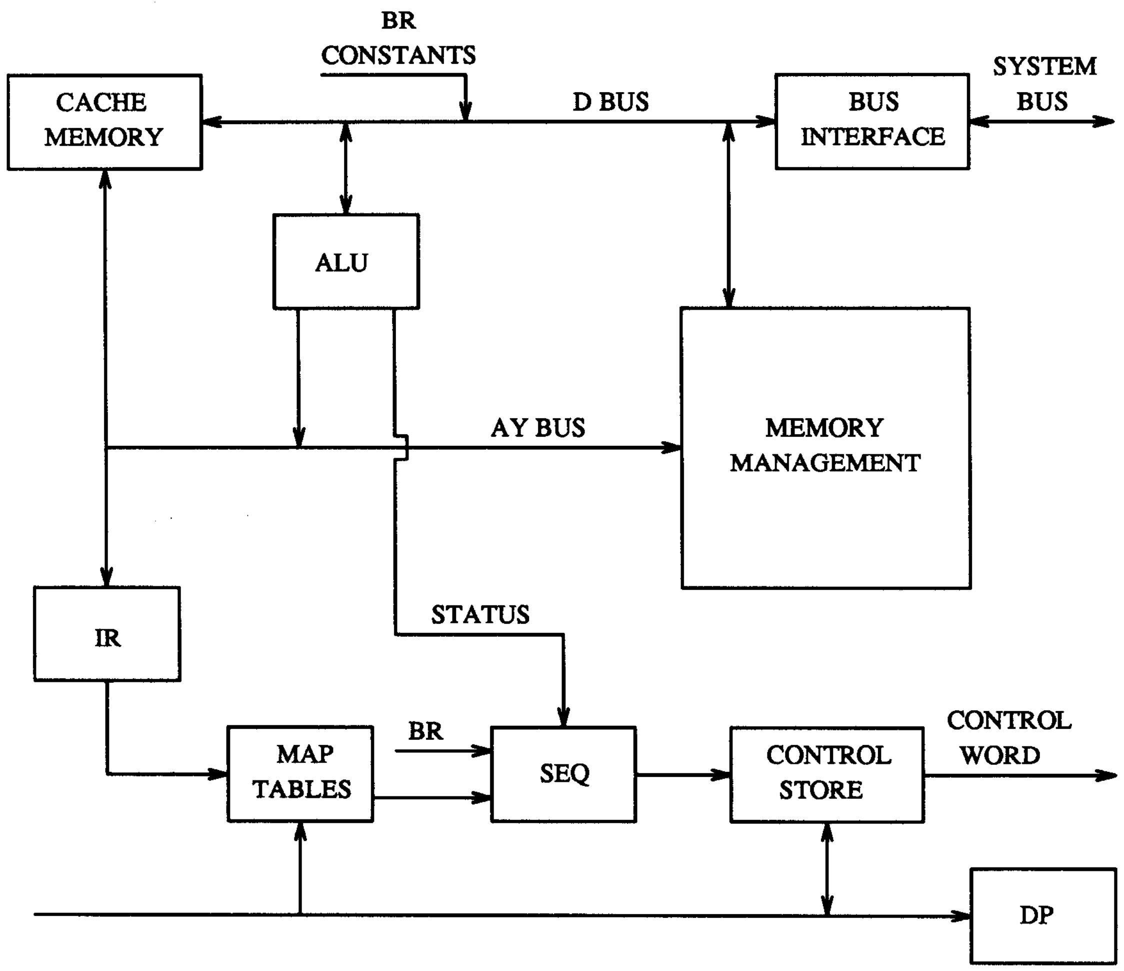

Figure 8.1: The Orion Central Processing Unit

8.2 The Orion Host Program

The source host program for the Orion Computer is given in Appendix F. This description was adapted from the BSPL specification of Richards [Richards 86].

8.2.1 Limitations of the Orion Host Program

The Orion BSPL specification, and consequently the host program, includes several simplifications to the actual Orion Computer.

1. It does not model the speed control facilities of the Orion [HLH 84]. Different microinstructions require varying amounts of time in which to execute. Two control bits in the control word are used to set the speed of the processor accordingly. The assembler gives a default setting which ensures correct operation, and a program called "speed" can be used to optimise the settings. The microprogrammer need not worry about the settings. The host program includes a local, `speed`, which selects the appropriate bits from the microword. However, this local is then not used.

2. The description does not take into account that the bus should not be locked for memory access for more than 8 microseconds [HLH 84]. The number of microoperations that this constitutes depends on the speed settings. Errors of this form would not be caught in any proof performed using the given host program. It could be modified to model this by including a fictitious state variable to keep a count of the time elapsed. Alternatively, this type of error could possibly be detected using dataflow analysis techniques.

3. It does not describe the Diagnostic Microprocessor (DP). This is an embedded microcomputer which is used both to bootstrap the Orion and perform diagnostic functions. It can also be used to dynamically load the control store and map tables. These cannot be written to by the Orion CPU itself, only by the DP. In the host program, they are modelled as state variables, `umem` and `map_mem`, which are assigned Z values on every cycle, and are thus left unchanged. All proofs using this host program include an implicit assumption that these state variables are not changed by the DP during the operation of the microprogram. It would perhaps have been more accurate to model them as being external to the CPU, representing them as inputs. An explicit requirement that they be unaffected would then have been necessary in any proof. The DP will also halt the CPU if a control store or map table parity error occurs. Again, this is not described in the host program.

   The CPU can communicate with the DP through a bytewide communication path. This is modelled by the input, `dp_inbyte`, and output, `dp_outbyte`. Two microoperations, ININT and OUTINT are used to interrupt the DP. Corresponding outputs are set to achieve this. Two status bits indicating the readyness of the DP are represented by inputs. Because the DP runs at a slower speed than the main CPU, the interrupt microoperations must be repeated. This is not described.

4. No IO-Subsystems are modelled. These contain microcomputers which remove the burden of much of the low level work involved in dealing with peripherals from the CPU. They are seen by the microprogrammer as memory mapped registers. In the host program, the whole of the available memory address space is dedicated to memory—a single state variable, `mem`, is used. There are no facilities for IO-Subsystems.

5. The Orion also provides Direct Memory Access (DMA) Channels for fast peripheral devices. A DMA request is honoured when the CPU next executes an `IDLE` bus microoperation. If a non-`IDLE` instruction is encountered, before the bus has been released, the CPU suspends itself. In the host program, the only way that memory can be changed is by CPU write operations. It is assumed that no DMA takes place. Any proof would be invalid in the presence of DMA.

Despite these simplifications, the host program used provides a realistic example of a commercial architecture. In particular it provides

- complexity due to the mass of detail,
- pipelines for increased performance and in particular pipelined microinstruction sequencing,
- potential unbroken feedback loops, and
- is largely horizontal in nature.

8.2.2 Converting the BSPL Step Function

As SPOOK originated from BSPL, converting the BSPL step function to a SPOOK host program was not difficult. It involved grouping rules with common subject together into single conditional assignments, ordering the assignments so that no variable was referenced before being used, isolating the potential feedback loop and making other minor syntactic changes.

Grouping Rules

The need to group together rules with a common subject into a single assignment had several ramifications. It made the specification more understandable in some respects, since in BSPL all the rules with a particular subject interact. All must be considered to determine the value associated with a variable. SPOOK forces single resources to be defined in a modular fashion. In general, this is very natural, and was largely how the BSPL specification was structured. In other situations, however, several rules are grouped under a particular guard such as when decoding vertical instruction fields like `mem_op`. Consider the BSPL rule

```
corrected_mem_op = 0                    //IDLE
    => { mar := #x_XXXX_XXXX
         prev_lrd, prev_rrd := #b00}
```

This is used in the Orion BSPL step function to give the action of an IDLE memory operation. The full action of such an operation can be easily identified. In the SPOOK host program, mar, prev_lrd and prev_rrd had to be given separate assignments. This means that while it is clear what the action on say mar is in any cycle, it is less obvious what the full effect of an IDLE operation is. In this respect the BSPL notation is more flexible.

Ordering Assignments

Ordering the assignments makes the specification more readable since it can be read sequentially. The assignment to any variable used will have occurred previously. In BSPL, the whole specification would have to be scanned to find the definitions of variables. This is very important with such large specifications if they are to be intellectually manageable. It has the disadvantage that the forced ordering will not necessarily correspond to an ordering of events that occurs in the actual hardware, where many updates to resources will occur in parallel. The host program over specifies the machine in this sense. This is not a great problem since the host program is only intended to have the same action as the machine from one clock tick to the next. It is not intended to be used to specify how those actions occur.

The Feedback Loop

One of the features of the SPOOK system is the simple way in which feedback loops unbroken by registers can easily be detected. The Orion Microarchitecture contains a potential feedback loop between the D bus and the AY bus through the shift and mask unit and the ALU as shown in Figure 8.2. The source host program contains a FEEDBACK command of size 2 to describe this situation. Each of the assignments to locals which represent wires that fall on this datapath are enclosed within the command.

```
FEEDBACK 2
 $( dbus      == ...ay...
    d_shifted == ...dbus...
    byte_3    == ...d_shifted...
    byte_2    == ...d_shifted...
    byte_1    == ...d_shifted...
    byte_0    == ...d_shifted...
    alu_data  == byte_3, byte_2, byte_1, byte_0
    r         == ...alu_data...
    rval      == ...r...
    f         == ...rval...
    ay        == ...f...   $)
```

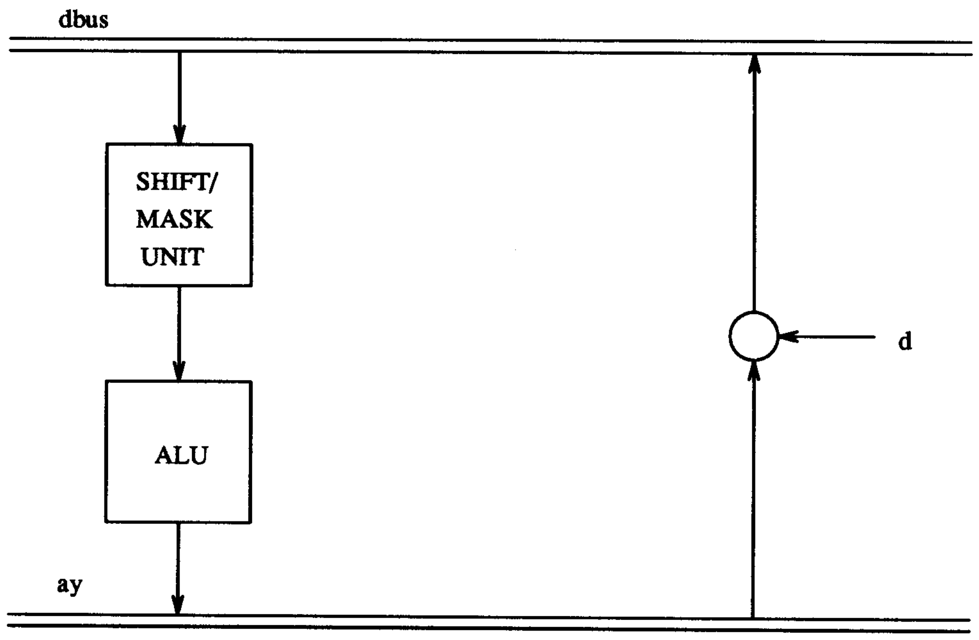

Figure 8.2: The Orion feedback loop

It expands to two copies of the body in the expanded host program.

The loop can be broken in several places by a multiplexor selecting a source from outside the loop.

- The D bus could take a non ALU value.
- The Mask unit could mask out all the values on the D bus.
- The ALU input could be from a source other than the shift and mask unit.
- The AY bus could take its value from the A input rather than the ALU.

Each such choice is indicated in the host program as a guarded or switch expression. The latter possibility, for example, is given by

```
ay == GUARDED
        || y_from_ay => reg_a
        DEFAULT f
```

The conditional expressions prevent U values from propagating when they occur in an unchosen clause.

If a particular microinstruction is such that the loop is not active, then generally the loop should collapse away. For example, if `y_from_ay` is known to be `#b1`, then the above assignment becomes

```
ay == reg_a
```

Host Program Version	SIZE
Source	607
Expanded	867
Preprocessed	792

Figure 8.3: The SIZE statistics for the Orion host program

A substitution will be created for this, causing the occurrence of **ay** in the second assignment to **dbus** to be replaced by **reg_a**. This assignment and the subsequent ones within the loop will then no longer be dependent on undefined locals, and their substitutions will be used instead of the earlier ones that did.

If the loop does not collapse away, it may be suggestive that the loop could be active if the microinstruction is executed in certain states. If the microprogram is guaranteed to work correctly then further information will be known about the state in which that microinstruction is executed. This information could be used in the collapsing process to remove the loop. The first indication that a loop may be active is that it does not collapse away.

8.2.3 Expanding and Preprocessing the Host Program

Expanding the Orion source host program follows the same pattern as for Gordon's computer and poses no great problems. The only notable difference is that the feedback loop must be dealt with. This simply involves replacing the loop by two copies of its body since the factor of the loop is 2. The expanding process involved increasing the SIZE of the source host program from 607 to 867. This difference is due to the feedback loop, checks, extra assignments for state variables and outputs and the extra default cases required for many conditionals.

The preprocessing which can be performed on the Orion expanded host program is very similar in nature to that which was done on the host program for Gordon's computer. In the preprocessing step the SIZE of the expanded host program is reduced by 9% as can be seen from Figure 8.3.

Many of the checks on the well-definedness of locals can be removed as the locals can be easily seen not to return U values. For the Orion host program there are many such checks (162) due to the large number of locals. Of these 46% are removed by the preprocessing. It is this check removal which is responsible for the reduction in the SIZE of the host program. The other preprocessing which is performed does not affect the SIZE.

Lookup tables frequently occur in the Orion host program and are represented by the concatenation of constants giving the values of the entries. This is done as it allows the contents of the lookup table to be formated and commented in the source host program. This situation is very similar to the expression assigned to **rom** in the host program of Gordon's computer. As there, the concatenation of constants can be evaluated to give a single constant at the preprocessing stage.

Again as with Gordon's computer there are places where a conditional expression is assigned to a local of smaller length in the source host program. In such situations, a selection operator is added during the expanding phase to truncate the expression. Such selections can be moved inside the conditional using the appropriate rewrite rule.

An additional occasion when expression rewriting can be performed is when a constant address is given in a subscript. This is sometimes used in preference to the equivalent selection. For example, in the Orion source host program, the expression

```
    ay !8 #b00
```

is used to refer to the lower byte of the AY bus. Such expressions are preprocessed to the selection form.

8.3 Control Flow and the Skeleton

8.3.1 Pipelined Instruction Fetching

One of the characteristic features of the Orion microarchitecture is its use of pipelining to allow the overlapping of functions which would otherwise have to be performed sequentially. In essence, the first function is calculated a cycle early, its value being held in a register. On the next cycle the second function can be performed immediately and at the same time as the execution of the first function for the subsequent cycle.

The most significant use of a pipeline is during the fetching of microinstructions to execute from the control store. A pipeline register (`instr`) holds the instruction which is being executed currently, and a microinstruction address register holds the address of the next instruction to be executed. This means that the next instruction can be executed immediately without having to first be fetched from the control store. This will always have been done on the previous cycle and the result stored in the pipeline register. The fetching of the next instruction, as indicated by the contents of the microinstruction address register, is performed in parallel with the execution of the current one. This complicates instruction sequencing, since, on each cycle, it is the address of the next but one instruction to be executed which is generated, rather than the next one. The address of the latter is already in the machine address register. This obscures the control flow in an assembler version of the code, especially when jumps are performed. A skeleton can be used to great effect to clarify the control flow. Consider the example below. Here, the instructions generate subsequent addresses, either by incrementing the machine address register, or by taking an address from a field in the control word. A `CJP` instruction performs a conditional jump to the given address if the given condition is true and otherwise generates sequential addresses. A `CONT` instruction just causes sequential addresses to be generated. The `NLC` condition code is the negation of the last condition code. It is true if the condition tested on the previous cycle was false and true otherwise.

```
CJP  ⟨ cc ⟩@1
CJP NLC @2
CONT

@1:     ⟨ label @1 code ⟩

@2:     ⟨ label @2 code ⟩
```

The above code executes both CJP instructions and then either the code at the location indicated by label @1, or the CONT followed by that at the location indicated by label @2 is executed. This can be seen much more clearly from the skeleton.

```
CASE  ⟨ condition corresponding to cc ⟩
  || #b1=>
        $(
           MI(#b1); // CJP  ⟨ cc ⟩@1
           MI(#b1); // CJP NLC @2
           ⟨ label @1 code ⟩
        $)
  DEFAULT
        $(
           MI(#b1); // CJP  ⟨ cc ⟩@1
           MI(#b1); // CJP NLC @2
           MI(#b1); // CONT
           ⟨ label @2 code ⟩
        $)
```

Since the microinstruction executed on each cycle is the one held in the pipeline register, it is that value which is used to produce collapsed host programs. Thus, MI commands will have the form

```
MI(SW instr .= value)
```

where *value* is the value of the actual control word to be executed. This differs from the skeleton for Gordon's computer, in which an address is specified.

8.3.2 The System Bus Control Function

Other state variables hold values which may radically alter the effect of the microinstruction, and so may be included in the MI conditions. One such state variable is connected with the system bus. It is via this bus that the physical memory and I/O subsystems are accessed. A pipeline is used when controlling the system bus. The required bus control function code must be transmitted to each device on the bus before it is used. For increased speed this code is transmitted a cycle in advance using three of the buses control lines. This means that the action of a microinstruction with respect to the system bus depends on the value transmitted on the previous cycle. This value is stored in the host program by a state variable, `mem_op`. The value of `mem_op` should in general be used in the collapsing process, i.e., given in the MI instructions.

8.3.3 ALU Function and Source Modifications

A further complication arises due to the limited facilities provided for a microinstruction to modify the effect of the next instruction to be executed. Functions are provided which cause the ALU function and source of the next instruction to be executed to be changed. If the ALU source modification was requested on the previous cycle, as represented in the host program by the state variable `ab_to_zb` having value `#s1`, then `AB` instructions (where one source is the A register and the other the B register) are converted to `ZB` (the first source becomes a zero constant), `AQ` to `ZQ`, `ZA` to `DQ` and `DA` to `DZ`. If the ALU function modification was requested, given by `div_ff` having value `#s1`, then `ADD` becomes `SUBR`, `SUBS` becomes `OR`, `AND` becomes `NOTRS` and `EXOR` becomes `EXNOR`. For instructions where the above modifications will have an effect, it is useful to include the value of the appropriate state variable in the MI condition.

8.3.4 The Alternative SIN Functions

The action of a microinstruction may also be affected by the previous microinstruction with respect to the SIN (serial in) microfield. The result of the ALU operation can be shifted by one bit before being sent to its destination. The SIN field in the microword determines what happens to the bits shifted in and out of the most and least significant bits when this occurs. There are two distinct sets of behaviour for this field—the "normal" set, which gives simple shifts and the "alternate" set, which gives more obscure functions used for special situations such as multiplication or division. A separate set of assembler mnemonics are provided for the alternate functions to enhance readability. However, the set of functions which is used does not depend on the SIN field, but must be requested by a special function in the previous microinstruction. This can be done explicitly using the ASIN special function, which just switches to the alternate set on the next cycle. It can also be done implicitly when certain other special functions such as QSAVE and TCDIV are requested. In the host program the special functions set the state variable `alt_set` to `#b1`. On the next cycle it is the value of this state variable that determines which set of SIN functions are consulted. Its value should be included in the MI condition if relevant.

A separate set of mnemonics was introduced in the assembler notation to help emphasise which function was actually being executed. However, as mentioned this does not force the appropriate function to actually be used and no check is made that the correct mnemonic is used. If the wrong one is given it could actually lead to confusion. In the skeleton, the fact that the alternate set was expected to be used would be indicated explicitly in the condition of the MI command. This would be checked during validation.

8.4 The Standard Instruction Set

The microcode implementing the standard instruction set provided with the Orion computer obeys several conventions about the way various registers in the register file are used and the state that the machine must be left in at the end of each macroinstruction. In this section, I describe those conventions which are relevant to the example considered in the remainder of this chapter. Instructions consist of a stream of bytes which are decoded by the microcode. Many are stack based and consist of a single byte opcode indicating the operation to be executed on the stack.

8.4.1 The Stack

A section of main memory is used by the standard instruction set to store the stack used by the stack based instructions. The stack pointer is held in one of the registers (`reg !32 #b_1110`). Part of the cache memory is used to store the top of this stack. The cache is a randomly accessed memory (`csh_mem`), which is split into banks—pairs of 512 word blocks. It is addressed using the cache address register which is split into three sections. The most significant four bits, or H section (`csh_h`), select the bank. The next most significant bit, or A section (`csh_a`), select the block within a bank pair. The least significant 9 bits, or L section (`csh_l`), select the word within the block. The top section of the stack is held in the normal block (`csh_a = #s0`) of one of the banks. The L section is then used as a 9 bit wrap round counter for the top of the stack. Its value is taken from the lower 9 bits of the actual stack pointer.

It is left to the microprogram to ensure that the stack does not overflow or underflow. This is done using a cache base register (`reg !32 #b_1011`) pointing to the lowest address held in the cache. The microcode for the procedure call and return instructions swap in or out part of the stack to make room for a new stack frame or bring in a previous one. The code for the expression evaluation instructions appear to assume that expression evaluation never overflows the stack, since they make no attempt to avoid overflow or underflow.

A pipeline is used with the cache memory. The value read from the code is buffered in a register (`csh_out`) so that the next value may be fetched while the current one is being used.

In the standard instruction set, the microcode implementing each instruction must conform to two conventions regarding the cache and the stack.

- All macroinstructions should leave the cache address register (modelled by the concatenation of the state variables `csh_h`, `csh_a` and `csh_l`) pointing to the top of the stack within the cache, as indicated by the stack pointer (R14). This is given by the condition

```
csh_h = #s0000 /\
csh_a = #s0 /\
csh_l = WS ((SW reg) !32 #b_1110)[8..0]
```

In addition the variable `csh_l` must be assured not to contain X signals, since it is used as part of the address to update locations in the cache memory. If the address is undefined an error will be caused. The predicate BINARYP can be used to assert this. It is a rather artificial assertion, since X signals do not correspond to real world objects. It is not possible to put an "X" value on a real wire—the value may be unknown but it will still be either a zero or one. In the real world this assertion can be thought of as specifying that the value of `csh_l` should not be random, since if that were so, an update to a random location would be made which is obviously undesirable. The following extra conjunct is added to the above condition

```
BINARYP(SW csh_l)
```

- The cache address register should not be changed by the last microinstruction of the code implementing an instruction. This ensures that the pipeline register (`csh_out`) holds the value at the top of the stack, which can then be read on the first microinstruction of the next instruction to be executed. This is given by the condition

```
csh_out =
   WS((SW csh_mem) !32 ((SW csh_h),(SW csh_a),(SW csh_l)))
```

The above conditions should be included in both the precondition and postcondition of the modules for any instruction.

8.4.2 Instruction Fetching

In the standard instruction set the microcode for each instruction must arrange to decode the opcode of the next instruction to be executed, and jump to the appropriate microinstruction. Instruction opcodes, and associated data, are fetched from memory eight bytes at a time and stored in two of the general purpose registers, R5 and R6. Assembler mnemonics *ir0* and *ir1* are set to correspond to these registers. In the SPOOK descriptions they are just part of the register file. For example, the expression, `(SW reg) !32 #b0101`, would give the contents of register *ir0* (R5). Register *ir0* is loaded with the instructions to be executed first. Bytes are then read into the lower byte of the instruction register (given by state variable `ir_l`) to be decoded except for control transfer instructions, which must discard the values in *ir0* and *ir1*. As bytes are read from *ir0*, it is shifted right to overwrite them, zeros being added at the most significant end. After four bytes have been read, the next opcode will be zero. This is reserved for an instruction which refills *ir0* (and *ir1* as necessary). The other instructions therefore need not do this refilling.

The opcode provides the lower bits of an address in the map tables which holds the address of the first microinstruction of the code implementing that opcode's instruction. The value at this address must be loaded into the instruction register `instr`. Also, the next address should be loaded into the microprogram address register (`umar`) and its successor into the 2910 microprogram counter (`pc`). The remainder of the address is given in the most significant bits of the instruction register, represented by the state variables ir_h and ir_a. A value of #s1 in ir_a indicates that the alternate set of instructions is to be used. This only occurs after an escape opcode. For other opcodes, ir_a has value #s0. Other instruction sets are specified by the value of ir_h.

The required post condition for non control transfer and non escape instructions is,

```
umar = (WS (Next_Ir_Address + 1)[11..0]) /\
pc = (WS (Next_Ir_Address + 2)[11..0]) /\
(SW reg) !32 #b0101 = Ir0 >> #b1000 /\
instr = WS ((SW umem) !64 (SW useg, Next_Ir_Address))
```

where Next_Ir_Address and Ir0 are ghosts defined in the precondition

```
Ir0 = (SW reg) !32 #b0101 /\
Next_Ir_Address = (SW map_mem) !15 (SW ir_h, #b0, Ir0[7..0])
```

8.5 The And Instruction

To illustrate the feasibility of the SPOOK system for proving Orion microcode, I will consider the implementation of the AND machine code instruction of the standard system. This code is short, consisting of only four microinstructions, though, due to the horizontal nature of the instructions and the underlying complexity of the Orion, it is not a trivial example. This instruction performs a bitwise AND operation of the top two words of the stack, removing the operands from the stack and adding the result to the top of the stack. The assembler version of this code, followed by a description of the code is given below.

```
ENTRY and
      CONT DZ SHR1 OR RAMA A=ir0 B=ir0 LDIR DECCA  (1)
      JUMP @9 DZ CSH OR RAMF B=R0                  (2)
      CJV DA CSH A=R0 AND RAMF B=R0                (3)
//- - - - - - -
@9:   CONT ZB SUBR RAMA A=R0 B=sp CWR              (4)
```

The code at address @9 is shared by several other instructions which perform similar stack operations. The first instruction decrements the least significant 9 bits of the cache address register as indicated by the function DECCA. The instruction register is also loaded from *ir0*, the bottom byte of which is then shifted out. To understand how this is done consult Figure 8.4 which gives a simplified version of the AM2901C. The RAMA function passes the A output of the register file onto the AY bus. The LDIR function then indicates that the lower byte of the AY bus is placed in the instruction register. This, therefore, has the effect of loading the lower byte of *ir0* into the instruction register. Now, the default D function is to connect the AY bus

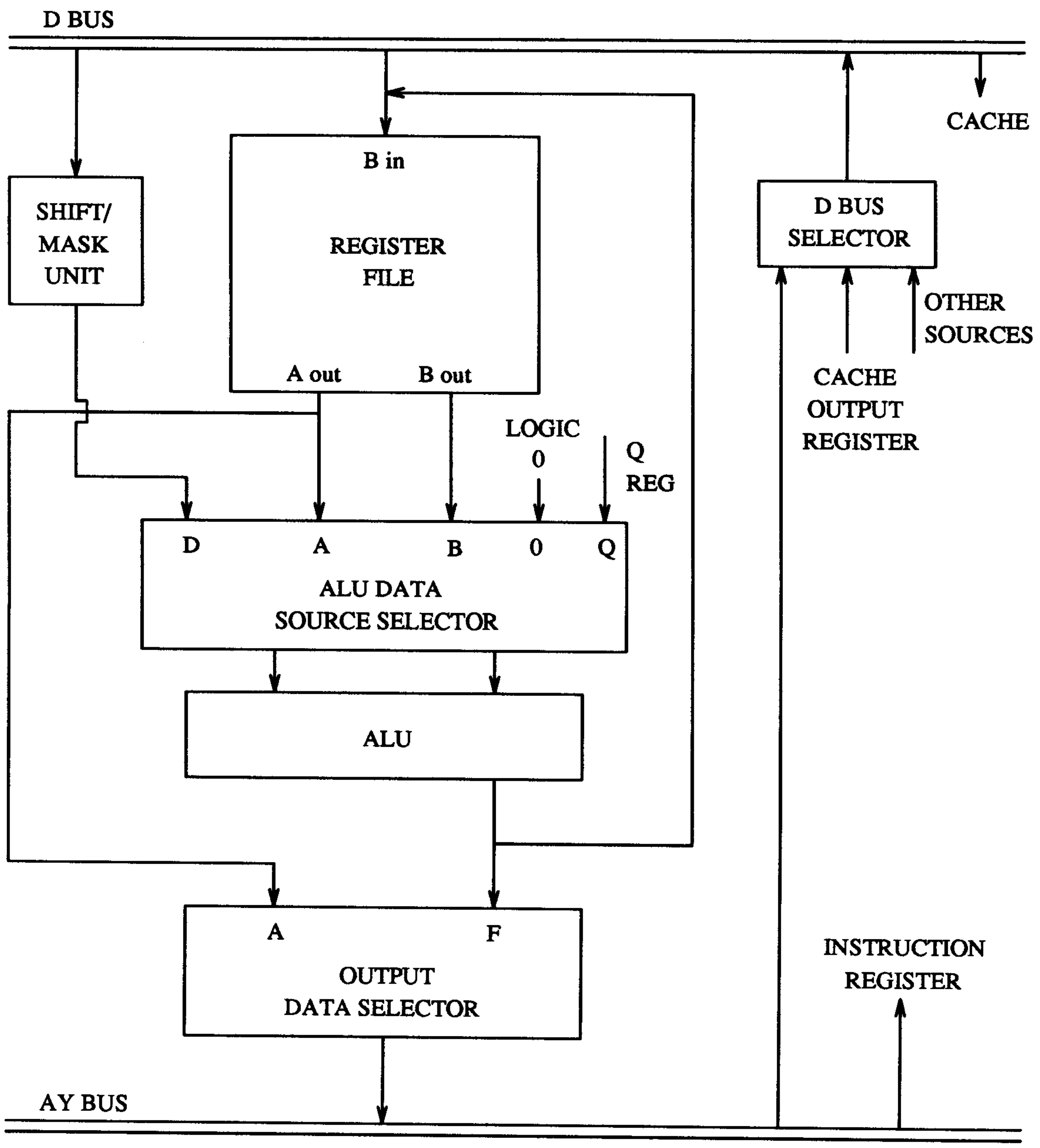

Figure 8.4: A simplified version of the AM2901C block diagram

to the D bus, so the value of *ir0* is also passed to the D bus. It then goes into the shift and mask unit where it is shifted right by one byte. The DZ function indicates that the value on the D bus and a zero argument are input to the ALU where an OR operation is performed. This has the result that the shifted contents of *ir0* is passed to the B input of the register file. This is set to *ir0*, achieving the desired result.

The value at the top of the stack is loaded into register R0 on the second instruction. The CSH function puts the contents of the cache output register (the top location of the stack) onto the D bus. This is the value at the location addressed *before* the first microinstruction decremented the cache address register, due to the pipeline. No shifting is performed on this value. It is then passed through the ALU as on the first instruction. The RAMF function connects the output from the ALU to the B input of the register file, which for this microinstruction is R0 so the top location of the stack is placed in R0. The JUMP @9 function makes a jump to the microinstuction at label @9. Due to the pipelined instruction sequencing, this jump is not performed until *after* the subsequent microinstruction.

The third instruction is similar to the second. The inputs to the ALU are taken from the D bus and the A output of the register file (DA). The A output is the contents of register R0 which holds the value from the top of the stack. The D bus takes its value from the cache output register, which now holds the value of the second item on the stack due to the decrement operation performed by the first microinstruction. An AND ALU operation is performed on the two arguments and the result placed back in register R0. The CJV function causes a jump to be made to the code to execute the next macroinstruction as indicated by the opcode in the instruction register. This jump is not made until after the next microinstruction has been executed.

The final instruction, at label @9, writes the value computed in register R0 back to the (new) top of stack. Register R0 is selected as the A output of the register file and passed directly to the AY bus using RAMA and subsequently to the D bus. The CWR function passes this value to the cache location given by the current contents of the cache address register—the new top of the stack. The ALU is meanwhile used to decrement the register used as the stack pointer (R14). This register is chosen as the B output of the register file and the ZB function is used to feed this and a zero into the ALU. SUBR performs a reverse subtraction (i.e., B − A as opposed to A − B). As no carry in is specified, an extra one is subtracted since the sense of carry in and out is reversed during subtraction. This value is fed back into the B input of the register file—the stack pointer—so the result is that the stack pointer is decremented. This keeps the stack pointer in step with the cache address register.

8.5.1 The Skeleton

The Skeleton module for the And code will have the form given below. The control flow through this module is sequential, the intricacies of the pipelined architecture having been avoided. The action of the microcode is much clearer.

```
MODULE And
  {...}
  $(
    MI (...) ; // CONT DZ...
    MI (...) ; // JUMP @9...
    MI (...) ; // CJV DA...
    MI (...) ; // CONT ZB...
  $)
  {...}
```

Since SPOOK uses the memory image of the microprogram, I assume that the four instructions are loaded contiguously from location zero for the purposes of this example. The hex version of the memory image considered is

```
0:  05590E0000135255
1:  1D011200300F5300
2:  1D010600000FEB00
3:  05050E00000E42E0
```

It is these values that the instruction register (`instr`) must hold at the start of each clock cycle.

The Conditions of the MI commands

The main information which is used to collapse the Orion host program is the value stored in the instruction register. For the first microinstruction of the And module, the following information can be used.

```
(SW instr) .= #x05590E0000135255
```

The memory operation to be performed provides further relevant information. Due to the pipeline delay in requesting memory operations, on the first microcycle of a macrocycle, the memory operation requested on the last microcycle of the previous macrocycle is executed. For example, a memory write requested at the end of a macrocycle would actually be executed during the next macrocycle. This means that the effect of the first microinstruction will depend on the macroinstruction just executed. For the purposes of this example, I will assume that the memory operation requested was IDLE—no memory operation was required. This may not always be a valid assumption. It simplifies the example, since all the assignments associated with memory operations are collapsed out. It is only sensible to issue lock, idle and write memory operations on the last cycle of a microinstruction, since read and address setting requests would perform no useful action. If this assumption were not made no great problems would be encountered in verifying the code. The first microinstruction would contain seven additional local assignments, together with corresponding check commands for four of the variables which would not collapse away. Since these assignments are fairly voluminous, the SIZE, in terms of the number of lines, of the collapsed host program would be increased approximately three fold. Each assignment would have a similar form, making a switch on a value at a position in a table dependent upon the value of **mem_op**. For example, the assignment to **mar'** would be:

```
mar'==
    SWITCH #b111...000 !3 (SW mem_op, (SW mar)[0])
     || #b000 => #bXX...XX
     || #b001 => (SW out_reg)[25..0]
     || #b101 => (SW out_reg)[25..0]
     DEFAULT Z_WORD 26
```

The assignments to the state variables **mar**, **prev_lrd** and **prev_rrd**, which are associated with three of these locals, are actually superfluous, since all are reassigned to in the subsequent microinstruction. They would be removed from the first instruction by the process of removing unused variable assignments. If the algorithm for local removal suggested in Section 5.5.3 were implemented, then the assignments to **prev_lrd'** and **prev_rrd'** could also be removed, since their checks would be collapsed out and their values would not be used.

To express the condition that the IDLE memory operation was expected, the following condition would be included in the condition for the first MI command.

```
(SW mem_op) .= #b000
```

To ensure that this condition does actually evaluate to **#b1**, a suitable condition must be included in the precondition of the module. The proof will only be valid when this precondition holds before execution of the module.

The first microinstruction should not be affected by ALU source and function modification since the **OR** function and **DZ** sources are used. The values of state variables **ab_to_zb** and **div_ff** therefore need not be used in the collapsing process. In fact, both should have value **#s0** since the previous macroinstruction should not be able to change the effect of a subsequent one in this way. If the information was required to determine the effect of the microinstruction, but not used, it would be apparent from the collapsed host program which would remain voluminous: none of the ALU code would be collapsed away. In states when the modification may or may not have been made on the previous instruction, the skeleton could be used to create two occurrences of the microinstruction; one when a modification had been made, and one when it had not. Similarly, the first microinstruction should not be affected by which **SIN** set is active, since it does not perform a shift on the ALU result. Whatever the value of **alt_set**, it will not affect the operation of the instruction or consequently aid the collapsing process. It need not be identified in the MI condition.

Other information that is known but which has only a minor effect on the collapsing process, is the next address information. Part of this information is held in the microprogram segment register (**useg**) which indicates which segment of the control store the microcode is held. In this example, I assume it to have value **#s000**. Also, the machine address register (**umar**) should hold the address of the next microinstruction (#s0000_0000_0001 in this case) and the 2910 microprogram counter (mpc) the next but one address. This information is given by the expression

```
(SW useg) .= #b000 &
(SW umar) .= #b0000_0000_0001 &
(SW pc)   .= #b0000_0000_0010
```

All of the above information is placed in the MI command corresponding to the first microinstruction.

```
MI((SW instr)  .= #x05590E0000135255 &
   (SW mem_op) .= #b000 &
   (SW useg)   .= #b000 &
   (SW umar)   .= #b0000_0000_0001 &
   (SW pc)     .= #b0000_0000_0010)
```

Similar information is used in the MI commands of the other instructions. For the second instruction, the values of the same variables: instr, mem_op, useg, umar and pc are known. When this instruction is executed the value of mem_op will definitely be #s000 since the memory IDLE function was requested by the first microinstruction. This will also be true for each of the other microinstructions. For the third microinstruction, additional information is required. It performs an ALU AND operation with source, DA. Both the function and source could be modified by the previous microinstruction. With only the information used to collapse the first two microinstructions, the operation of the ALU would not be fully determined, and the amount of collapsing possible seriously reduced. Since it is known that the previous microinstruction is not intended to make such modifications, this information, notably that the variables ab_to_zb and div_ff have value #s0, could be used to perform the collapsing. For the final microinstruction, only the values of instr and mem_op are known and useful. None of the address information is known because a CJV instruction was just executed. The addresses depend on the next macroinstruction to be executed. The full body of the skeleton is given in Figure 8.5.

8.5.2 Expanding the Skeleton

The Collapsed Host Programs

The collapsed host programs which result from using the information in the skeleton are a significant improvement on the voluminous expanded host program. The SIZE statistics for the collapsed host programs are given in Figure 8.6. These statistics take account of the check resulting from the MI condition in addition to the actual collapsed host program. They show that the expanded host program is collapsed by 93% for each instruction. This compares with the degree of collapsing performed for Gordon's computer (92%), illustrating that the collapsing process scales well to complex architectures.

```
SKELETON
{...}
 $(
   // CONT DZ SHR1 OR RAMA A=ir0 B=ir0 DECCA LDIR
   MI(SW instr    .= #x05590E0000135255 &
      SW mem_op   .= #b000 &
      SW useg     .= #b000 &
      SW umar     .= #b000000_000001 &
      SW pc       .= #b000000_000010);

   // JUMP @9 DZ CSH OR RAMF B=R0
   MI(SW instr    .= #x1D011200300F5300 &
      SW mem_op   .= #b000 &
      SW useg     .= #b000 &
      SW umar     .= #b000000_000010 &
      SW pc       .= #b000000_000011);

   // CJV DA CSH A=R0 AND RAMF B=R0
   MI(SW instr    .= #x1D010600000FEB00 &
      SW mem_op   .= #b000 &
      SW ab_to_zb .= #b0 &
      SW div_ff   .= #b0 &
      SW useg     .= #b000 &
      SW umar     .= #b000000_000011 &
      SW pc       .= #b000000_000100);

   // @9: CONT ZB SUBR RAMA A=R0 B=sp CWR
   MI(SW instr    .= #x05050E00000E42E0 &
      SW mem_op   .= #b000)
 $)
{...}
```

Figure 8.5: The body of the skeleton for the Orion AND instruction

Microinstruction	SIZE	Reduction from Expanded Host Program (Original SIZE 867)
1	63	93%
2	60	93%
3	62	93%
4	62	93%

Figure 8.6: The SIZE Statistics for the Orion And Collapsed Host Programs

The collapsed host program for the first microinstruction is given in Figure 8.7. Some locals remain. In Figure 8.7 the full details of these assignments are omitted. They are given in full in Figure 8.9. They each involve guarded expressions and so were considered to be too complex to collapse forwards during the collapsing process. The guarded expressions themselves could not be simplified further because the guards involve input values. The first three assignments are responsible for setting various parity bits and have the form,

```
parity == GUARDED
          || GET(set_parity; time) => #b1
          DEFAULT Z_WORD 1
```

The remaining local assignments include guards having the form

```
#b0 & GET(...; time)
```

This expression cannot be simplified to `#b0` allowing the guard to be removed, since the get expression could return a Z value corresponding to a high impedance signal being placed in the corresponding input port. If that happened, the result of the expression would be `#bU`. Since this would ultimately cause an error, the external environment must ensure that the relevant inputs are never given Z values. This information would be included in the precondition. The information is not available to the collapsing algorithm and so cannot be used to make the above simplification. If it were included in the MI condition, it would be possible for the algorithm to make use of it, collapsing out the assignments to both `ir_1'` and `dp_outbyte'`, though this has not been implemented.

The only check commands that remain are for the well-definedness of the locals whose assignments were not collapsed away. Since these locals are only guaranteed to be well-defined under the assumptions discussed above, even if they were collapsed forwards with no further simplification, the checks would not be removable. This would only be possible if the additional information were used to perform the collapsing. All the other checks have been removed, indicating that all other locals are guaranteed to be given well-defined values. No errors due to bad values occurring in the combinatorial logic can occur, nor can any race conditions. Also, no state storing devices have been created other than the state variables. In particular, the feedback loop has been completely collapsed away and all the checks on its locals have been removed. For the microinstruction considered the feedback loop cannot be active.

The host program is collapsed to a similar extent for each of the microinstructions given appropriate information in the conditions of the MI commands.

8.5.3 Unused Variable Assignment Removal

Unused variable removal proves to be extremely useful for the Orion as can be seen from the SIZE statistics of Figure 8.8. For each of the first three microinstructions, the corresponding collapsed host program can be reduced by approximately a further 30%. No unused assignments can be removed from the last microinstruction's

```
BLOCK
 $(
  csh_parity'   == (GUARDED...);
  tb_parity'    == (GUARDED...);
  mem_parity'   == (GUARDED...);
  ir_l'         == (GUARDED...);
  dp_outbyte'   == (GUARDED...);
  CHECK (WD dp_outbyte');
  CHECK (WD ir_l');
  CHECK (WD mem_parity');
  CHECK (WD tb_parity');
  CHECK (WD csh_parity');
  dp_outbyte    == PUT(dp_outbyte; time; dp_outbyte');
  outint        == PUT(outint; time; #b_0);
  inint         == PUT(inint; time; #b_0);
  PAR
   $(
    pc          := #s_00000000_0011
    last_cc     := #s_1
    prev_lrd    := #s_0
    prev_rrd    := #s_0
    mar         := #s_XXXXXXXX_XXXXXXXX_XXXXXXXX_XX
    mem_op      := #s_000
    scff        := #s_0
    mem_parity  := MERGE(mem_parity; mem_parity')
    csh_parity  := MERGE(csh_parity; csh_parity')
    qr_save_bit := #s_X
    csh_out     := WS((SW csh_mem) !32 ((SW csh_h),(SW csh_a),(SW csh_l)))
    c_save_bit  := #s_0
    div_ff      := #s_0
    reg         := MERGE(reg; #b_0101 != (#b_00000000,(SW reg)[191..168]))
    ab_to_zb    := #s_0
    raw_wflt    := WS((SW tb_parity) | ...)
    tb_parity   := MERGE(tb_parity; tb_parity')
    raw_rflt    := WS((SW tb_parity) | ...)
    ir_l        := MERGE(ir_l; ir_l')
    ir_a        := #s_0
    csh_l       := WS(((SW csh_l) - #b_00000000_1)[8..0])
    alt_set     := #s_0
    umar        := #s_00000000_0010
    useg        := #s_000
    instr       := WS((SW umem)[127..64])
    time        := INC time
   $)
 $)
```

Figure 8.7: The collapsed host program for microinstruction 1 of the And module

Microinstruction	SIZE	Reduction from Expanded Host Program (Original SIZE 867)	Reduction from Collapsed Host Program (Original SIZE in brackets)	
1	44	95%	30%	(63)
2	43	95%	28%	(60)
3	44	95%	29%	(62)
4	62	93%	0%	(62)

Figure 8.8: The SIZE statistics for the Orion And collapsed host programs with unused assignments removed

collapsed host program because all could be used subsequently, i.e., be referred to in the post condition of the module.

The collapsed host program with unused assignments removed for the first microinstruction is given in Figure 8.9. Due to the Orion computer's complex architecture, many registers are frequently side-effected values that are not used. It contains numerous flags, holding state information which could be tested on the subsequent microinstruction, but which in the And implementation are not. Some, such as the last condition code flag, `last_cc`, could be selected as the condition code which is used by the microprogram controller. When they are not selected, the assignment to these flags can be removed. The memory read and write fault flags are possible candidates for the condition code. It is advantageous for their assignments to be removed, since the expressions assigned to them after collapsing are fairly complex. Other flags record status information about whether the next instruction is to be modified. For example, the flag `ab_to_zb` indicates whether the ALU source of the next instruction is to be modified. Often the value is not actually required and so the assignment can be removed.

In addition to the single bit flags, some full registers are often candidates for unused variable assignment removal. One such register is the main memory address register (`mar`) which is given an undefined (X) value on cycles which request `IDLE` memory operations. This occurs for all of the microinstructions of the And instruction. These assignments must always be removable, since if they were not a memory request would be being made for an unspecified address. Assignments to the cache pipeline register (`csh_out`) will frequently be removable since it is updated on every cycle whether the value is used or not. Since the And instruction performs a stack operation it reads from the cache and so only the assignment to `csh_out` in the third microinstruction can be removed.

From the collapsed host program with unused variable assignments removed, the precise relevant action of the microinstruction can be seen. The informal documentation for the first instruction states only that, "the instruction register is reloaded". This corresponds to the assignment

```
    ir_1 := MERGE(ir_1; ir_1')
```

Provided that the input `inrdy` is non Z as discussed above, `ir_1'` will have the value of `(SW reg)[167..160]` so `ir_1` will be loaded with bits 160 to 167 from the

```
BLOCK
 $(
   csh_parity'  == (GUARDED
                     || GET(set_csh_parity; time) => #b_1
                     DEFAULT Z_WORD 1);
   tb_parity'   == (GUARDED
                     || GET(set_tb_parity; time) => #b_1
                     DEFAULT Z_WORD 1);
   mem_parity'  == (GUARDED
                     || GET(set_mem_parity; time) => #b_1
                     DEFAULT Z_WORD 1);
   ir_1'        == (GUARDED
                     || #b_0 & GET(inrdy; time) => GET(dp_inbyte; time)
                     || #b_1 => (SW reg)[167..160]
                     DEFAULT Z_WORD 8);
   dp_outbyte'  == (GUARDED
                     || #b_0 & GET(outrdy; time) => SW(ir_1)
                     DEFAULT Z_WORD 8);
   CHECK (WD dp_outbyte');
   CHECK (WD ir_1');
   CHECK (WD mem_parity');
   CHECK (WD tb_parity');
   CHECK (WD csh_parity');
   dp_outbyte   == PUT(dp_outbyte; time; dp_outbyte');
   outint       == PUT(outint; time; #b_0);
   inint        == PUT(inint; time; #b_0);
   PAR
    $(
     pc         := #s_00000000_0011
     mem_op     := #s_000
     mem_parity := MERGE(mem_parity; mem_parity')
     csh_parity := MERGE(csh_parity; csh_parity')
     csh_out    := WS((SW csh_mem) !32 ((SW csh_h),(SW csh_a),(SW csh_1)))
     reg        := MERGE(reg; #b_0101 != #b_00000000,(SW reg)[191..168]))
     tb_parity  := MERGE(tb_parity; tb_parity')
     ir_1       := MERGE(ir_1; ir_1')
     ir_a       := #s_0
     csh_1      := WS(((SW csh_1) - #b_00000000_1)[8..0])
     umar       := #s_00000000_0010
     useg       := #s_000
     instr      := WS((SW umem)[127..64])
     time       := INC time
    $)
 $)
```

Figure 8.9: The collapsed host program with unused assignments removed for the first microinstruction of the And module

register file. This corresponds to the lower 8 bits of register R5. In addition, register R5 is rotated by one byte, losing the byte that has been loaded into the instruction register.

```
    reg := MERGE(reg; #b0101 != (#b00000000,(SW reg)[191..168]))
```

The A section of the instruction register is also cleared.

```
    ir_a := #s000
```

It can be seen from the collapsed host program with unused assignments removed that other actions occur:

- the next instruction is loaded into the instruction register (`instr`) and assignments are made to the microaddress registers (`pc`, `umar` and `useg`);
- the parity bits could be set;
- the cache pipeline register is loaded from the cache with the value at the top of the stack;
- an idle memory operation is initiated;
- the least significant 9 bits of the cache address register (`csh_l`) is decremented in readiness for the second stack item to be loaded, and
- the outputs which indicate to the Diagnostic Processor that a byte has been accepted (`inint`) and that a byte is ready (`outint`) are cleared, provided that an error is not caused due to the Diagnostic Processor "ready to receive" input being given a floating value.

8.5.4 The Target Specification

The Precondition

As indicated in the discussion of the standard instruction set, several conditions concerning the cache memory must hold immediately prior to the execution of any standard instruction set instruction. These are gathered into the definition of a single predicate `CACHE_OK`. This predicate must be included in the precondition of the And skeleton. Also, the conditions described earlier, giving the values of ghost variables concerned with instruction fetching, must be included in the preconditions of all standard instruction set instructions including And. These conditions are given by the predicate `SET_IR_GHOSTS`.

Since the And instruction manipulates the stack, other ghost variables are set which record the values of the various stack associated variables at the start of the macrocycle. The ghost variable, `Stack`, records the value of the whole of the cache memory.

```
    Stack = SW csh_mem
```

The name of this variable, "Stack", is misleading since the stack itself resides only in part of the cache memory—a single block of one bank. Also, it is only the top

```
    Tos = Stack !32 (#b00000, Sp[8..0]) /\
    Nos = Stack !32 (#b00000, (Sp[8..0] - 1)[8..0])
```

These conditions are given by the predicate SET_STACK_GHOSTS in the precondition. Similar predicates would be required for all standard instruction set instructions which manipulate the stack.

As an IDLE memory operation was assumed to have been requested for the first microinstruction when expanding the skeleton, this must also be included in the precondition. This is done using the predicate IDLE_MEM_OP.

```
    |- IDLE_MEM_OP (mem_op) = mem_op = #s000
```

A further predicate, AND_MEM is used to indicate the required state of the control memory before the And instruction is executed. This states that the control memory does hold the four microinstructions of the code for the And macroinstruction.

```
    |- AND_MEM (umem) =
            SW umem !64 #o00000 = #x05590E0000135255
            SW umem !64 #o00001 = #x1D011200300F5300
            SW umem !64 #o00002 = #x1D010600000FEB00
            SW umem !64 #o00003 = #x05050E00000E42E0
```

Orion microprograms cannot modify themselves so if this condition holds prior to the execution of the code, then those instructions will be executed. However, if this were not so and these microinstructions were modified, the check on the MI command's condition would cause abortion to occur, preventing the proof from succeeding. A proof that no unwanted self modification of the control memory occurs is an integral part of the verification, and of any other form of validation attempt. Furthermore, if the control memory was intentionally modified, then provided the correct instructions ultimately reside in the instruction register at the appropriate time, the proof will still go through as required.

Restrictions are required on the values of many of the input variables, and these must be included in the preconditions. As indicated earlier if the inputs used as guards have non binary (#b0 or #b1) values at a particular time then an error will occur. The predicate INPUT_BINARYP is used in a similar fashion to the way it was used in the target specification of Gordon's computer. The necessary conditions are grouped together in the predicate INPUT_RESTRICTIONS.

```
|- INPUT_RESTRICTIONS (...) =
       INPUT_BINARYP (outrdy) /\
       INPUT_BINARYP (inrdy) /\
       INPUT_BINARYP (set_mem_parity) /\
       INPUT_BINARYP (set_tb_parity) /\
       INPUT_BINARYP (set_csh_parity)
```

Finally, the values in the various microinstruction address registers and instruction register must be specified. The instruction register should hold the first microinstruction of the And code.

```
instr = #x05590E0000135255
```

The segment register should hold value #s000, since the And code is assumed to be in that segment.

```
useg = #s000
```

The two address registers umar and pc should point to locations 1 and 2 respectively, since the And code is assumed to be in the lower 4 memory words of the segment for the purposes of this example.

```
umar = #s000000_000001 /\
pc   = #s000000_000010
```

The Postcondition

According to the Orion Manual, the And instruction replaces, "NOS and TOS by NOS ⟨ op ⟩ TOS", where TOS is the top location of the stack, NOS is the next location and ⟨ op ⟩is a bitwise *and* operation. This must be specified in the postcondition of the And instruction. The ghosts set up in the precondition may be used to refer to the initial values of the resources. The bitwise *and* operation can be specified using the SPOOK & operator and this should be performed on the original values on the top of the stack, given by the ghost variables Nos and Tos. The address to be updated is that in the stack part of the cache memory at the location where the second stack item was stored, i.e.,

```
#b00000, (Sp - 1)[8..0]
```

The merge and subscription update operators can be used to describe the original value of the cache memory updated at a particular location.

```
csh_mem = MERGE(WS Stack;
                    (#b00000, (Sp - 1)[8..0]) != (Nos & Tos))
```

This also specifies that no other locations in the cache memory are affected. The stack pointer must also be decremented to point to the new top of stack.

```
SW reg !32 #b1110 = (Sp - 1)[31..0]
```

These conditions are given by the predicate **AND**.

No mention is made in this specification about underflow of the portion of the [illegible] memory or other stack related registers. Such restrictions must be included in the specifications of each instruction.

As with all instructions, the And instruction must obey the same conventions about the cache as were included in the precondition, and given by the predicate CACHE_OK. Also, the code should obey the conventions of instruction fetching detailed in Section 8.4.2. These conventions are given by the predicate IR_FETCHED.

Finally, the predicate NO_ALU_MOD is included. This asserts that no ALU function or source modification is requested for the next cycle.

```
|- NO_ALU_MOD (ab_to_zb; div_ff) =
       ab_to_zb = #s0 /\
       div_ff   = #s0
```

The full postcondition has the form

```
{ AND(...) /\
  CACHE_OK (...) /\
  IR_FETCHED (...) /\
  NO_ALU_MOD (...)
}
```

It is this specification that the proof shows that the code is consistent with. For the purposes of this example, some relevant information has been omitted. Other conditions are required for a complete proof. The extra information would not increase the difficulty of the task in any great way, and would add nothing to the discussion. For example, the specification does not state that the other registers in the register file are not altered. The first five registers are scratch registers and so never carry meaningful values between macroinstructions. The others, however, are conventionally used to store useful information. For example, register R9 is used as the program status word, storing the likes of the interrupt disable bit. Such registers should obviously not be corrupted by the high level instructions. Other conventions may also be assumed by the other instructions in the standard instruction set, such as that the alternate SIN set is not chosen in the last microinstruction of a macrocycle. Furthermore, as indicated earlier, it was assumed that an IDLE memory operation was requested for the first microinstruction. The proof is only valid when this assumption is true. This highlights the point that a proof is only as good as the specification upon which it is based.

Source	GOAL_COUNT	GOAL_COUNT left	GOAL_COUNT proved
Well-defined checks	21	1	95%
MI condition checks	4	0	100%
AND predicate	2	1	50%
CACHE_OK predicate	5	2	60%
IR_FETCHED predicate	4	3	25%
NO_ALU_MOD predicate	2	0	100%
Total	38	7	81%

Figure 8.10: The GOAL_COUNT statistics for the Orion And module

Annotations

Since the skeleton of the And instruction consists simply of a sequence of four microinstructions, no annotations are required within it.

8.6 Verification Conditions

The extra complexity of the Orion Microarchitecture provides no great difficulty when generating verification conditions, since the host program still has the same simple structure.

Figure 8.10 shows that the GOAL_COUNT for the And module is 38. The majority (21) of this arises from the unremoved well-defined checks from the four microinstructions – all but one of which being from locals which were not collapsed forwards. These were all proved automatically by the AUTO_THM tactic. These checks had not been removed prior to the verification condition generation because the information about the binaryness of certain inputs which was required was not available to the collapsing process. This information was provided by the precondition, and so was picked up by the AUTO_THM tactic. If a mechanism for placing and extracting it from the the MI conditions had been implemented, these checks would have been removed during the collapsing process, and would not have appeared in the GOAL_COUNT statistics. This illustrates that the more successful the collapsing process, the less favourable the GOAL_COUNT statistics will appear, as the simple things are done at an earlier stage.

The one well-defined check which was not proved resulted from the assignment to csh_mem' in microinstruction 4.

```
csh_mem' ==
    ((SW csh_h), (SW csh_a), (SW csh_l)) != (sw reg)[31..0]
```

This results in an unproved goal of the form

```
{...} |- IS_ONEP (WD ((SW csh_h), (SW csh_a),
                      ((SW csh_l) - #b00000000_1)[8..0]) != ...
```

The subscription update operator is defined to return U values if its first argument returns X values. This means that to prove the above goal involves determining that the expression,

```
    (SW csh_1) - #b00000000_1
```

does not return X values when `csh_1` does not. Only a few rules about X analysis were provided to the system, and so it does not contain the above knowledge. Thus, the goal could not be proved automatically.

A GOAL_COUNT of 4 arises from the checks resulting from the MI conditions – one for each MI command. These were all proved automatically. Their proof indicates that the control structure suggested by the skeleton does match that of the actual microcode and that the instructions expected to be executed are executed. Clutterbuck [Clutterbuck 86] made use of "shallow proofs" in his work on proving the correctness of machine code programs to perform proofs that just certain dynamic errors did not occur. The situation is analogous to the checks in the SPOOK system. Similar "shallow proofs" could be performed to show that the skeleton structure is adhered to or that no errors occur due to locals being given U values and in particular that no feedback loops are active.

The GOAL_COUNT generated from the predicate `NO_ALU_MOD` was proved completely automatically. This consisted of proving two simple goals of the form

```
    {...} |- #s0 = #s0
```

If extra conditions had been added to the postcondition stating that the contents of various registers had not been corrupted by the And code as discussed earlier, the resulting goals would have been of a similar simple nature to the above and so easily proved automatically.

The `AUTO_THM` tactic was not as successful in proving the goals corresponding to the other predicates, as they are slightly more complex. Their truth can easily be seen and their automatic proof within the system would become possible simply by adding a few new expression rewrite rules. For example, a typical unproved goal resulting from the predicate `CACHE_OK` is,

```
    {...} |-
          WS(((SW reg)[456..448] - #b00000000_1)[8..0]) =
          WS((#b11111111_11111111_11111111_11111111 +
              (SW reg)[479..448])[8..0])
```

By converting the subtraction operator to its equivalent plus form and using the commutativity of plus and constant evaluation, this would become

```
    {...} |-
          WS((#b11111111_1 + (SW reg)[456..448])[8..0]) =
          WS((#b11111111_11111111_11111111_11111111 +
              (SW reg)[479..448])[8..0])
```

The outer selection can then be moved inside the addition on the right hand side of this equality to give the goal which with constant evaluation and the use of the rewrite rule, EXP.23(Select-select), can trivially be seen to be true.

Each of the new rules used in the above argument could be easily represented in the system as rewrite rules, allowing AUTO_THM to prove the goal. Furthermore, the same rules are required in the proofs of most of the other unproved goals, and all would appear to be useful in the proofs of other architectures. The same rules are required in the proof of some of the unproved goals of Gordon's computer.

Overall 81% of the goals required to be proved were proved by the AUTO_THM tactic. This compares well with the figure of 86% for the verification of Gordon's Computer, once more indicating that the techniques scale up well to the more complex architecture. The comparison would have been even better had the target specification been more complete, including information about state variables that should not have been modified by the code, since these would generate trivial goals.

8.6.1 Errors Found

No errors were found in the microcode of the And Instruction by the verification process. This is not surprising since the code is a very short and commonly used piece of code. Also, the designers of the system were not available to be questioned about the intended behaviour of the instruction. The specification is therefore more a specification of what the code actually does rather than of what the designers intended it to do.

Two errors were found in the target specification due to unprovable goals being generated. The first was that the post condition originally specified that umar should hold the address of the next instruction to be executed.

```
umar = WS Next_Ir_Address
```

This is incorrect, since the next microword to be executed should already reside in the instruction register instr and so umar should point to the next location.

```
umar = ((WS Next_Ir_Address) + #b00000000_0001)[11..0]
```

The unprovable goal had the form

```
{...} |-
      (WS Next_Ir_Address) =
      ((WS Next_Ir_Address) + #b00000000_0001)[11..0]
```

The second error was similar. The microprogram counter (pc) was specified to hold a value one less than it actually did. The unprovable goal was similar to that above.

An actual memory image was verified. This means that any bit errors introduced, for example, by the assembler could have been detected by the verification process.

If the memory image had been peppered with bit errors, only those which affected the code in such a way that it no longer matched the specification would have been detected. Often, the values of fields in the control word are not relevant. For example, the values in the fields specifying the A and B outputs from the register file are not always used in a microinstruction, so bit changes to them will not be detected by the verification. Also, if the specification is inadequate, and does not specify conditions that are actually important, errors which violate these conditions would not be detected. For example, corruptions to the values in some important registers in the register file would not have been detected by the above verification since the condition that such registers were not altered was not included in the specification.

Chapter 9

Epilogue

9.1 Summary

One of the major obstacles to proving microcode for production systems is in handling the complexity of detail. This occurs in two forms:

- complexity due to proving properties about the large bodies of microcode needed to implement production macroarchitectures, and

- complexity due to the myriad of detail involved in the execution of single microinstructions for real microarchitectures.

In this dissertation I have presented ways of tackling both these problems.

In the software world the former problem has been tackled by advocating the use of structured, modular programming languages. It is clear that validation has a greater chance of success if the code is broken into intellectually manageable pieces, and the constructs used have simple and clean semantics. However, microprogramming is most naturally done at a low level. Even if universal high level microprogramming languages supported by verification technology became widely available, some low level coding would still be performed: despite the fact that conventional high level languages have long been available much machine code programming is done. In this dissertation, I have shown that the advantages of structured, modular programming can be obtained without the code being written that way. Instead, the structure and division can be given in the documentation in the form of a skeleton. Consequently, the inefficiencies and insecurities of high level programming are avoided, and the programmer does not have to write the code with verification in mind in this way.

The second problem, of the complexity of detail within the microarchitecture, is overcome using collapsed host programs in conjunction with the skeleton. This not only reduces the complexity at an early stage, but also provides a useful tool to help understand the action of both individual microinstructions and the microprogram as a whole. This was illustrated using code for the Orion computer. Whilst only a small fragment of code was considered it demonstrated the feasibility of tackling real microarchitectures.

The tools would be of use whether the validation was performed *post facto* by an independent verification team, or hand in hand with the proof, by the programmer. The use of the skeleton itself would be an advantage in the former respect, since it provides a formal validatable way in which the programmer could convey the intended control structure of the microprogram to the verification team. Also, using the collapsing tools would help provide these teams gain insight into the way the code worked. The tools would also be useful whilst writing the code. It is generally accepted that formal verification alone is not sufficient to give complete confidence about the correctness of code, but should be used in conjunction with other validation techniques such as testing. The ideas presented here could easily be incorporated into an integrated validation system, and indeed would be advantageous to such a system as a whole. For example, an interpreter for SPOOK would provide a universal tool for simulating microprograms. Also, if used on the expanded skeleton, it would potentially provide a fast simulator, since much of the work in simulating the host is removed by the collapsing process. In addition, the collapsed host programs and expanded skeleton are human readable providing an additional validation technique—checking by eye that the code is as expected.

On the whole, only very simple algorithms were used: the theorem prover was very naive; more could be done at the assignment removal stage; more information could have been extracted from the MI conditions when collapsing, etc. Despite this, much was achieved. If the algorithms were enhanced in the ways suggested, much better performance could be obtained in more general cases.

9.1.1 A Comparison of the Case Studies

Gordon's computer is a simple "toy" design so the fact that verification techniques can handle its complexity does not imply that they will provide usable tools for production architectures. The Orion microarchitecture is a production architecture and is significantly more complex than Gordon's computer. This is indicated to some extent by the SIZE statistics for the expanded host programs. For the Orion the SIZE is approximately six times larger than for Gordon's computer. Furthermore, the latter's host program includes the whole of the control ROM contents in this figure. If this is not included, the Orion is nearer eight times larger. Since the Orion is a production computer, with this increased size also comes added complexity in the form of pipelining, feedback loops, lookup tables etc.

Only a small fragment of code was considered for the Orion though this is enough to suggest whether the increased complexity in the host prevents the techniques from scaling well. On the whole, the results were very encouraging, and are suggestive that the techniques do provide useful and usable tools for the documentation and validation of microprograms, and that they do scale well to complex architectures. The collapsing process was extremely useful for the Orion simply as a tool to facilitate understanding of the semantics of the complex microinstructions, especially since the official informal documentation was scant. Leaving the advantages with respect to formal verification aside, this in itself is sufficient motivation for using the techniques described.

The statistics collected were similar for the two case studies. Less check removal could be performed in the preprocessing stage on the Orion expanded host program – only 46% of the check commands could be removed in comparison with 70% for Gordon's computer. This corresponds to a 9% reduction in the size of the Orion expanded host program as opposed to a 15% reduction for Gordon's computer. Therefore, the preprocessing was not quite as effective in this way for the Orion. However, other preprocessing which did not affect the SIZE statistics such as constant evaluation was performed for the Orion. The preprocessing was still a useful technique to employ.

The full collapsing process did scale extremely well to the Orion. In both examples, the expanded host programs were collapsed by 92% to 93% on average. Furthermore, for the Orion, additional simplification could be performed to great effect by removing unused assignments.

In both case studies, the vast majority of the goals requiring proof were of a very trivial nature. Even though only very simple and naive theorem proving tools were used, 81% of the goals could be proven automatically for the Orion And microcode and 86% for the code for Gordon's computer. All the goals that were not proved automatically were simple and could conceivably have been proved automatically. Often an overlap in the additional rewrite rules and tactics required occurred.

It was apparent in both case studies that an important technique in the collapsing and theorem proving stages was the ability to determine when expressions could not return non-binary values. This was a prerequisite for the application of many rewrite rules and in particular for dealing with the well-defined checks in both the collapsing and theorem proving stages. The simple techniques used for this task appeared to be satisfactory. For the well-defined checks, much of the analysis could be performed in the preprocessing stage to great effect.

9.1.2 Collapsed Host Programs

One of the main techniques used was that of collapsing the host program. This overcomes the problem of the complexity of the underlying detail and has the additional advantage of giving the user a concise description of the exact effect of individual microinstructions. The technique proved to be very useful and successful, scaling up well to the real microarchitecture of the Orion Computer. To a large extent this success was due to the very simple structure of the expanded host program, allowing a process of constant folding and rewriting to be used. The process was hindered to some extent by the presence of X, Z and U signals, rather than just binary ones. This meant that analysis of whether expressions could return these signals was required before rewrites were applicable. The analysis was largely straightforward, however, and so this was not a significant problem.

A potential problem of the collapsing process is that the rewrite rule set used is not sufficient to competently collapse the host programs for other microarchitectures. The collection of rules was fairly ad hoc and was used on only a single commercial example. It is probable that for other microarchitectures, other rules would be useful. However, in general, the host programs for other architectures will be written in a similar style so the bulk of the collapsing will be done by just a few rules, notably those for the conditionals, selection and subscription. It seems likely that sufficient collapsing would still be performed.

The heuristic used to temper the collapsing process, notably that of not collapsing assignments of conditional expressions forwards appears to work well. The resulting collapsed host programs were both concise and readable, even when insufficient information was provided to seed the collapsing. When this occurred it meant that microinstruction was dependent upon more information than the skeleton suggested. This served to highlight such dependencies and make the user consider more carefully the state in which the microinstruction was to be executed.

9.1.3 The Skeleton

The skeleton forms the crux of the techniques used. Previous attempts to convert the firmware verification problem into a software verification one have involved verifying code with complex control structures, full of jumps. By using the skeleton, well-structured code results, making the verification task more tractable. Since microprograms generally implement short numeric algorithms, the resulting code is of a similar nature to the simple algorithms that have formed the majority of the examples used in software verification research.

Providing a skeleton was very natural, since this type of information is often given informally as a state transition diagram, and is an effective method for making code more understandable. It brings many advantages. It allows the code to be modularised, overcoming the problems of verifying large microprograms. It provides the information used to collapse the host program, allowing the complexities due to the microarchitecture to be removed at an early stage. Using a skeleton also reaps other advantages. It is validatable documentation and is a convenient framework upon which to place assertions. It is also a suitable way for a microprogrammer to convey information about the microprogram to a verification team. The expanded skeleton provides a concise, formal and understandable version of the microprogram upon which simulations can be performed.

9.1.4 The Model of the Microarchitecture

The model of the microarchitecture as a clocked finite state machine, though simple, was sufficient to describe the Orion Microarchitecture, a significant real example. By treating the host in this way, the division of a microinstruction into microoperations was avoided. In production architectures, such a division is rarely clear cut and introduces problems of side effects. Furthermore, with such a model complex proof rules concerning the interactions of different microoperations acting in parallel are required. Avoiding these problems by keeping the semantics of the host language simple facilitated the automation of both the proof and collapsing process: a necessity if the tools are to be used with production architectures. One problem which was solved in a fairly simple way was that of identifying the presence of feedback. This problem does not appear to have been given great attention in the in the literature. In the SPOOK system, such active loops are identified in the normal course of the verification, due to the well-defined check commands. The same technique also prevents state storing devices such as flip flops, other than those declared as state variables, from being created. The way this was to be done for BSPL, where a similar problem exists, was not addressed in the BSPL report [Richards 86].

The method used for describing the host had some disadvantages. Since no facilities for giving detailed timing characteristics were provided, verification at this level could not be performed. The model used for inputs assumed that input signals occurred in discrete time units, as given by the microclock. For the duration of the microcycle, inputs were assumed to hold a single value which changed on the clock tick. In general, this would not be so—the values could change at any time. Errors due to unclocked inputs of this form would not be detected. Also, the sequential nature of the host program overspecifies the microarchitecture in that it gives an ordering to internal events. This is not a disadvantage given the way that the specification was used, since we were only interested in the state at the boundaries of the clock cycle. It would be a disadvantage if the host program had been intended as a specification from which to implement the hardware. A further disadvantage is the lack of modularity in the host program. More modularity is obtained than with BSPL in that all the 'rules' concerning a single resource are packaged together within a single assignment. The structure of the host is however flat. This facilitates the generation of verification conditions but is of little help to the person writing the host program and to others reading it. It would be better in this respect if it were more hierarchical in nature.

9.1.5 Theorem Proving

Semi-automatic theorem proving tools were used within the system. This approach appeared to work well, since the majority of the tedious work could be performed automatically, while still affording the user some degree of control. When the presence of errors was detected in the microprogram or specifications, this made it much easier to discover their location and correct them. In a fully automated system, if errors were discovered due to unprovable goals, their cause would be much less obvious to the user. Systems such as the Boyer-Moore theorem prover produce voluminous output describing exactly what the prover was doing. Whilst being a help, this leaves the user with the task of wading through a large amount of text to determine what was going wrong.

Many of the unproved goals in the case studies were of a very simple nature and could have been proven within the system had extra rewrite rules been added. It is not unforeseeable, for example, that the Orion And instruction code be proven completely automatically. Tailoring the system to a particular example in this way would not necessarily have given a good picture of its capabilities, however. If a different example were considered, it is likely that different problem domain dependent rules would be required. To some extent the system was tailored in this way, though only rules which appeared to be generally useful were added. The most natural way to overcome this problem would be to allow the user to provide additional rules and tactics specific to the problem in hand. This has the danger that the user might carelessly add invalid rules purely because they were what was apparently needed to prove a particular goal. It is especially easy to forget the possibility of expressions returning X, Z and U values. It would only be sensible to allow this if the actual proof steps were checked as is done in the HOL style systems for example.

9.1.6 The Advantages of the SPOOK System

As suggested in Chapter 2, the SPOOK system gains many of the advantages of each of the other types of microcode verification systems. No particular programming practices are forced on the microprogrammer. The code may be produced in any way; the low level code may be changed at the last minute and still be verified, and *post facto* verification is a possibility. The system is universal and not restricted or tailored to any particular microarchitecture. If the microarchitecture evolves during the design cycle, then the tools are capable of evolving with it. The tools will also be usable for small projects where it would not be feasible to produce a whole new set of tools. Simulators could be used on the expanded skeleton, allowing the possibility of a fast, universal testing framework. The documentation, in the form of the skeleton, would be validated at the same time as the microprogram. An additional validation technique is provided for—checking the various versions of the code by eye. The action of individual microinstructions can be inspected using collapsed host programs, which give a concise picture of what a microinstruction does. The way microoperations interact and any side effects occurring will be explicit in this representation. Also, the expanded skeleton gives a human readable and accurate version of the microprogram, which itself can be inspected for errors. This is of more use than simply disassembling the code since it gives greater and more accurate detail than an assembler notation whilst omitting irrelevant detail. The microcode is treated as a program and not data. This allows the proof to be modularised according to the problem. The structure does not have to be rediscovered in the course of the proof. The user is made to provide additional formal documentation of the system. The way the microcode implements the target is documented in addition to documenting the target itself. This is useful if the microprogram is to be modified. In particular, the skeleton gives an untangled and consequently easier to understand version of the state transition diagram of the microprogram. It also helps with *post facto* verification. This documentation is validated in conjunction with the microprogram itself.

9.1.7 Discussion

Much discussion has occurred in the literature as to the usefulness of formal verification (see for example [De Millo 79] [Fetzer 88] [Barwise 89]). The result of the verification process is to show that the microprogram in conjunction with the host program is consistent with its partial correctness specification. This does not necessarily mean that the program is "correct". To this end, it is perhaps more useful to think of the specification being documentation of what the code does, rather than a specification of what it is intended to do. What it is intended to do is just a notion in the head of the designer. Writing the specification involves attempting to capture this notion. Specifications are just models which may or may not give sufficiently accurate models of the real world. Consider the Orion host program. This only gives a simplified description of the actual hardware—direct memory access and the diagnostic microprocessor are not modelled, for example.

Furthermore, if the formal specification of the target is written after the microprogram has been completed as with the examples presented here, there is a great danger that the verifier will write a specification with the code in mind, mirroring mistakes in the code in the specification. An additional way that specifications can be inadequate is if they are too weak. In the Orion And specification, it was not stated that the unused registers in the register file were not altered. It would have been an error if certain of these registers had been altered by the code. Since the condition was not included in the specification, the verification did not show that the code was "incorrect". Such conditions are easy to neglect. In a similar way the verification of Gordon's computer does not consider whether the computer powers up correctly. If the specifier does not think of a potential problem then its absence may not be verified.

This does not mean that formal verification is pointless, however. The aim of the exercise is to obtain more reliable systems. If errors are found by the verification as in the two case studies described, then the verification was worthwhile. Even if these errors were in the specifications it is useful to have found them, since accurate documentation is important if the code is to be used correctly or subsequently modified. Occasionally errors could be found more easily by other methods. For this reason, a range of validation techniques should be used. The important thing is that the errors be found as early as possible in the design cycle, especially for microprograms, where mistakes become increasingly expensive to correct.

9.2 Further Work

9.2.1 An Integrated Validation System

This dissertation has been primarily concerned with the formal verification of code. Other validation tools could also be incorporated into the system and gain advantage from the techniques described. The most obvious such tool would be a simulator. Furthermore, only partial correctness has been considered. Termination is also an important correctness property of code which should be considered. As with the proof of partial correctness, the termination proof could be carried out on the expanded skeleton gaining similar advantages.

9.2.2 Examples Considered

The system has only been used to verify a small amount of code. These examples have shown the potential of the system. Further examples are needed to fully explore this potential.

All the examples considered involved *post facto* verification of existing code. Despite the fact that this is considered more difficult than verifying code whilst writing it, I have illustrated its feasibility using the system, and that the system provides advantages in this direction. However, the tools could also be used during the development process. Further work, could be performed using the system in this way.

Code for the Orion computer was used to illustrate the feasibility of using the system for real microarchitectures. Only a small fragment of code was verified, however. The verification of large bodies of real code would probably raise new problems, due to the increased size of the task. The proof of Gordon's computer illustrated the way such a proof could be modularised, reducing the problem to many smaller and intellectually manageable ones. It seems promising that the system would scale up to such tasks. The proof of a large body of real code would be needed to illustrate this.

It would also be useful to apply the system to a wider range of microarchitectures. It was designed specifically with the Orion in mind. The rewrite rules used were to some extent tailored to that machine. It would be interesting to see if the rules are sufficiently flexible and widely applicable to completely collapse other host programs. If this were not so, then other rules would need to be developed. By tailoring the rules to collapse the underlying structures being modelled, such as look up tables, it seems likely that they could be useful generally.

A major correctness problem when dealing with microcode that is held in a writeable control store is that of ensuring that it does not erroneously modify itself. The SPOOK system would appear to have potential in this direction, since each MI instruction would include information about the expected contents of the location to be executed. This need not be the original contents of the location. Any such information would be automatically verified as part of the proof, so it would seem that such proofs would naturally be performed as part of the verification. Experimentation is needed to illustrate this potential.

Since the model of the microarchitecture used is that of a finite state machine, the universality of the system is not restricted to microprograms. It could be used to verify properties of anything that could be modelled in this way. In particular it could be used to verify machine code programs. Also, the top level specification of the VIPER microprocessor is modelled as a finite state machine, so proofs of its properties could conceivably be carried out.

9.2.3 Theorem Proving

The theorem proving facilities provided are not very powerful. Most verification conditions are of a very simple nature and can easily be proved with such a simplistic prover. The few that cannot must be proved by some other means. Current state of the art theorem provers such as HOL and the Boyer-Moore system could provide mechanical aid. This would involve building suitable theories of SPOOK expressions within them.

9.2.4 Security

The SPOOK system as currently implemented is insecure. There is no guarantee that the rewrite rules and tactics implemented are correct, and therefore that the theorems proved are actually true. This contrasts the systems such as HOL, which use a type checking mechanism to ensure that any theorem can only be accepted as such if it has been derived from a sequence of applications of primitive inference rules and axioms. Some work has been performed in mechanising programming logics within theorem proving systems such as HOL. This suggest that it may be feasible to implement SPOOK within such a system. This would involve defining a denotational semantics for the language. From this the axiomatic semantics and transformational rules would be derived. Processes such as the collapsing of the host program would then involve applying suitable tactics. An additional advantage would be that the full power of the theorem prover would be available when proving the verification conditions.

9.2.5 Extending the Language

The SPOOK language is fairly simple, containing few commands. To allow the control constructs appearing in microprograms to be more naturally modelled, the commands available could be extended, providing, for example, different loop constructs. Only constructs amenable to verification ought to be included, however. The module facility provides a very simple way of partition the code and illustrates the way this could be done. For complex architectures a more powerful procedure call mechanism would probably be needed, allowing the modularised code to be shared. This would especially be so if the microarchitecture itself had a procedure call facility. One other area where the language is lacking is the ability for the user to define functions for use in the host program. Whilst the predefined functions provided are useful for most purposes it seems likely that other more specialised functions would be needed. This must be remedied.

Bibliography

The pages on which each reference is cited are listed in parentheses after the reference.

[Barrow 84a] Harry G. Barrow. Proving the Correctness of Digital Hardware Designs. *VLSI Design*, 64–77, July 1984. (15)

[Barrow 84b] Harry G. Barrow. VERIFY: A Program for Proving the Correctness of Digital Hardware Designs. *Artificial Intelligence*, 24:437–491, 1984. (15, 65, 123)

[Barwise 89] Jon Barwise. Mathematical Proofs of Computer System Correctness. *Notices of the American Mathematical Society*, 36(7):844–851, September 1989. (204)

[Berg 81] Helmut K. Berg. Firmware Testing and Test Data Selection. In *Proceedings of the National Computer Conference*, 1981. (2)

[Bergeretti 85] Jean-Francois Bergeretti and Bernard A. Carré. Information-Flow and Data-Flow Analysis of While-Programs. *ACM Transactions on Programming Languages and Systems*, 7(1):37–61, January 1985. (2)

[Birman 74] Alexander Birman. On Proving Correctness of Microprograms. *IBM Journal of Research and Development*, 250–266, May 1974. (3, 12)

[Birman 76] Alexander Birman and William H. Joyner, Jr. A Problem Reduction Approach to Proving Simulation Between Programs. *IEEE Transactions on Software Engineering*, SE-2(2):87–96, June 1976. (12)

[Blikle 76] A. Blikle and S. Budkowski. Certification of Microprograms by an Algebraic Method. In *Proceedings of the 9th Annual Microprogramming Workshop*, pages 9–14, 1976. (14)

[Boyer 79] R. S. Boyer and J. S. Moore. *A Computational Logic.* Academic Press, New York, 1979. (15)

[Brand 78] Daniel Brand and William H. Joyner, Jr. Verification of Protocols using Symbolic Execution. *Computer Networks*, 2(4/5):351–360, 1978. (12)

[Brand 82] Daniel Brand and William H. Joyner, Jr. Verification of HDLC. *IEEE Transactions on Communications*, Com-30(5):1136–1142, 1982. (12)

[Brioschi 76] A. Brioschi and P. Scaini. The Microprogram's Correctness. In *Euromicro 76*, North-Holland Publishing Company, 1976. (17)

[Budkowski 78] S. Budkowski and P. Dembiński. Firmware versus Software Verification. In *Proceedings of the 11th Annual Microprogramming Workshop*, pages 119–127, 1978. Appeared as a special issue of *Sigmicro Newsletter*, 9(4), December 1978. (13, 88)

[Camilleri 86] Albert J. Camilleri, Michael J. C. Gordon, and Tom Melham. *Hardware Verification Using Higher Order Logic.* Technical Report 91, University of Cambridge, Computer Laboratory, September 1986. (14)

[Camilleri 88] Albert J. Camilleri. *Executing Behavioural Definitions in Higher Order Logic.* PhD thesis, University of Cambridge, Computer Laboratory, July 1988. Also appeared as University of Cambridge, Computer Laboratory Technical Report 140. (15)

[Carré 86] B. A. Carré, I. M. O'Neill, D. L. Clutterbuck, and C. W. Debney. SPADE – The Southampton Program Analysis and Development Environment. In Ian Sommerville, editor, *Software Engineering Environments*, pages 129–134, *IEE Computing Series 7*, Peter Peregrinus, 1986. (2)

[Carter 75] William C. Carter, William H. Joyner, Jr., and George B. Leeman, Jr. Automated Experiments in Validating Microprograms. In *Proceedings of the Fault Tolerant Computing Symposium*, pages 51–55, Volume 5, July 1975. (3, 12)

[Carter 77] William C. Carter, H. A. Ellozy, William H. Joyner, Jr., and George B. Leeman, Jr. Techniques for Microprogram Validation. *Agardograph*, (224):9-1–9-19, April 1977. (12)

[Carter 78a] William C. Carter, William H. Joyner, Jr., and Daniel Brand. Microprogram Verification Considered Necessary. In Saki P. Ghosh and Leonard Y. Liu, editors, *AFIPS Conference Proceedings 1978 National Computer Conference*, pages 657–664, Volume 47, AFIPS Press, June 1978. (3, 12)

[Carter 78b] William C. Carter, William H. Joyner, Jr., Daniel Brand, H. A. Ellozy, and J. L. Wolf. An Improved System to Verify Assembled Programs. In *Proceedings of the Fault Tolerant Computing Symposium*, pages 165–170, IEEE Computer Society, Volume 8, June 1978. (12)

[Carter 79] William C. Carter, Daniel Brand, and William H. Joyner, Jr. Symbolic Simulation for Correct Machine Design. In *Proceedings of the 16th Design Automation Conference*, pages 280–286, IEEE Press, June 1979. (12)

[Clint 73] M. Clint. Program Proving: coroutines. *Acta Informatica*, 2:50–63, 1973. (48, 94)

[Clint 84] M. Clint and C. Vicent. The Use of Ghost Variables and Virtual Programming in the Documentation and Verification of Programs. *Software Practice and Experience*, 14(8):711–737, August 1984. (48)

[Clutterbuck 86] D. L. Clutterbuck. *The Validation and Verification of Low Level Code.* PhD thesis, Department of Electronics and Computer Science, University of Southampton, 1986. (18, 105, 195)

[Clutterbuck 88] D. L. Clutterbuck and B. A. Carré. The Verification of Low Level Code. *Software Engineering Journal*, 97–111, 1988. (18, 105)

[Cohn 87] Avra Cohn. *A Proof of Correctness of the Viper Microprocessor:The first Level.* Technical Report 104, University of Cambridge, Computer Laboratory, January 1987. (3, 14, 88)

[Cohn 88] Avra Cohn. *Correctness Properties of the Viper Block Model.* Technical Report 134, University of Cambridge, Computer Laboratory, 1988. (14)

[Crocker 80] Stephen D. Crocker, Leo Marcus, and Dono van-Mierop. The ISI Microcode Verification System. In G. Chroust and J. P. Mühlbacher, editors, *Firmware, Microprogramming and Restructurable Hardware*, pages 89–103, North-Holland Publishing Company, 1980. (13)

[Crocker 88] Stephen D. Crocker, Eve Cohen, Sue Landauer, and Hilarie Orman. Reverification of a Microprocessor. In *Proceedings of the 1988 IEEE Symposium on Security and Privacy*, pages 166–176, Computer Society Press of the IEEE, April 1988. (3, 13, 88)

[Cullyer 88] W. J. Cullyer. Implementing Safety Critical Systems: The Viper Microprocessor. In Graham Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, chapter 1, pages 1–25, Kluwer Academic Publishers, 1988. (14)

[Damm 84a] Werner Damm. Automatic Generation of Simulation Tools: A Case Study in the Design of a Retargetable Firmware Development System. In B. Myhrhaug and D. R. Wilson, editors, *Advances in Microprocessing and Microprogramming (10th Symposium)*, pages 165 – 176, Elsevier Science Publishers B.V. (North-Holland), 1984. (10)

[Damm 84b] Werner Damm. An Axiomatization of Low Level Parallelism in Microarchitectures. In *Proceedings of the 17th Annual Microprogramming Workshop*, pages 314–323, 1984. Appeared as a special issue of *Sigmicro Newsletter*, 15(4), December 1984. (10)

[Damm 85a] Werner Damm. Design and Specification of Microprogrammed Computer Architectures. In *Proceedings of the 18th Annual Microprogramming Workshop*, pages 3–9, 1985. Appeared as a special issue of *Sigmicro Newsletter*, 16(4), December 1985. (10)

[Damm 85b] Werner Damm and Gert Döhmen. Verification of Microprogrammed Computer Architectures in the S* System: A Case Study. In *Proceedings of the 18th Annual Microprogramming Workshop*, pages 61–73, 1985. Appeared as a special issue of *Sigmicro Newsletter*, 16(4), December 1985. (3, 10)

[Damm 86] Werner Damm, Gert Döhmen, Klaus Merkel, and Mathilde Sichelschmidt. The AADL/S* Approach to Firmware Design Verification. *IEEE Software*, 3(4):27–37, July 1986. (10)

[Damm 88] Werner Damm. A Microprogramming Logic. *IEEE Transactions on Software Engineering*, 14(5):559–574, May 1988. (10, 164)

[Dasgupta 80] Subrata Dasgupta. Some Implications of Programming Methodology for Microprogramming Language Design. In G. Chroust and J. R. Mühlbacher, editors, *Firmware, Microprogramming and Restructurable Hardware*, pages 243–252, North-Holland Publishing Company, 1980. (9)

[Dasgupta 81] Subrata Dasgupta. S^*_A: A Language for Describing Computer Architectures. In M.Breuer and R. Hartenstein, editors, *Computer Hardware Description Languages and their Applications*, pages 65–78, North-Holland Publishing Company, 1981. (9)

[Dasgupta 84] Subrata Dasgupta and Alan Wagner. The Use of Hoare Logic in the Verification of Horizontal Microprograms. *International Journal of Computer and Information Sciences*, 13(6):461–490, 1984. (9)

[Dasgupta 86] Subrata Dasgupta, Philip A. Wilsey, and Juha Heinanen. Axiomatic Specifications in Firmware Development Systems. *IEEE Software*, 3(4):49–58, July 1986. (9)

[Dasgupta 88] Subrata Dasgupta. Principles of Firmware Verification. In Stanley Habib, editor, *Microprogramming and Firmware Engineering Methods*, chapter 10, pages 433–482, Van Nostrand Reinhold, 1988. (7)

[Davie 88] Bruce S. Davie. *A Formal, Hierarchical Design and Validation Methodology for VLSI.* PhD thesis, University of Edinburgh, Department of Computer Science, October 1988. (123)

[De Millo 79] R. A. De Millo, R. J. Lipton, and A. J. Perks. Social Processes and Proofs of Theorems and Programs. *Communications of the ACM*, 22, 1979. (204)

[Dembiński 78a] P. Dembiński and S.Budkowski. An Introduction to the Verification Oriented Microprogramming Language "MIDDLE". In *Proceedings of the 11th Annual Microprogramming Workshop*, pages 139–143, 1978. Appeared as a special issue of *Sigmicro Newsletter*, 9(4), December 1978. (13)

[Dembiński 78b] P. Dembiński and S.Budkowski. Verification, Design and Description Oriented Microprogramming Language. In H. W. Lawson, H. Berndt, and G. Hermanson, editors, *Proceedings of the Conference on Large Scale Integration – Technology, Applications and Impacts*, pages 230–240, EUROMICRO 78, North-Holland Publishing Company, 1978. (13)

[Dembiński 83] P. Dembiński. Design and Verification Oriented Microprogram Transformations. In J. Boormann, editor, *Proceedings of the IFIP Conference on Programming Languages and System Design*, pages 141–155, IFIP, Elsevier Science Publishers B. V. (North-Holland), 1983. (14)

[Fetzer 88] James H. Fetzer. Program Verification: The Very Idea. *Communications of the ACM*, 31(9):1048–1063, September 1988. (204)

[Floyd 67] Robert W. Floyd. Assigning Meanings to Programs. In *Proceedings of the Symposium on Applied Mathematics*, pages 19–32, Volume 19, 1967. (18, 96)

[Foster 86] J. M. Foster. Formally Based Static Analysis of Microcode. In *Proceedings of the 19th Annual Microprogramming Workshop*, 1986. Appeared as a special issue of *Sigmicro Newsletter*, 17(4), December 1986. (2)

[Gordon 79] Michael J. C. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: a Mechanised Logic for Computation. Lecture Notes in Computer Science, 78*, Springer–Verlag, 1979. (12, 14)

[Gordon 81a] Michael J. C. Gordon. *LCF-LSM.* Technical Report 41, University of Cambridge, Computer Laboratory, 1981. (14)

[Gordon 81b] Michael J. C. Gordon. *Proving a Computer Correct.* Technical Report 42, University of Cambridge, Computer Laboratory, 1981. (14, 23, 123, 163)

[Gordon 87] Michael J. C. Gordon. *A Proof Generating System for Higher Order Logic.* Technical Report 103, University of Cambridge, Computer Laboratory, 1987. (14)

[Gordon 88] Michael J. C. Gordon. *Mechanizing Programming Logics in Higher Order Logic.* Technical Report 145, University of Cambridge, Computer Laboratory, 1988. (112)

[Gries 81] David Gries. *The Science of Programming.* Springer-Verlag, 1981. (10)

[Hanna 86] F. K. Hanna and N. Daeche. Specification and Verification of Systems using higher-order predicate logic. *IEE Proceedings*, 133(E-5):242–254, September 1986. (14)

[HLH 84] *ORION Microarchitecture Reference Manual.* High Level Hardware Limited, Windmill Road, Oxford, 2nd edition, 1984. (1, 23, 169)

[Hoare 69] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580,583, October 1969. (22, 109)

[Hunt 85] Warren A. Hunt, Jr. *FM8501: A Verified Microprocessor.* Technical Report, University of Texas, Austin, dec 1985. (15, 88)

[Igarashi 75] Shigeru Igarashi, Ralph L. London, and David C. Luckham. Automatic Program Verification I: A Logical Basis and its Implementation. *Acta Informatica*, 4(2):145–182, March 1975. (8)

[Joyce 86] Jeffrey J. Joyce, Graham Birtwistle, and Michael J. C. Gordon. *Proving a Computer Correct in Higher Order Logic.* Technical Report 100, University of Cambridge, Computer Laboratory, December 1986. (14, 123, 163)

[Joyce 88a] Jeffrey J. Joyce. *Formal Specification and Verification of Microprocessor Systems.* Technical Report 147, University of Cambridge, Computer Laboratory, September 1988. (14)

[Joyce 88b] Jeffrey J. Joyce. Formal Verification and Implementation of a Microprocessor. In Graham Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, chapter 4, pages 129–151, Kluwer Academic Publishers, 1988. (14, 90)

[Joyce 89a] Jeffrey J. Joyce. Formal Specification and Verification of Microprocessor Systems. *Integration, the VLSI Journal*, 7(3):247–266, September 1989. (14)

[Joyce 89b] Jeffrey J. Joyce. *Multi-Level Verification of Microprocessor-Based Systems.* PhD thesis, University of Cambridge, Computer Laboratory, 1989. (14)

[Joyce 89c] Jeffrey J. Joyce. *A Verified Compiler for a Verified Microprocessor.* Technical Report 167, University of Cambridge, Computer Laboratory, March 1989. (8)

[Joyner 76] William H. Joyner, Jr., William C. Carter, and George B. Leeman, Jr. Automated Proofs of Program Correctness. In *Proceedings of the 9th Annual Microprogramming Workshop*, pages 51–55, 1976. Appeared as a special issue of *Sigmicro Newsletter*, 7(3), September 1976. (12)

[Joyner 78] William H. Joyner, Jr., William C. Carter, and Daniel Brand. Using Machine Descriptions in Program Verification. In J. Moneta, editor, *Information Technology: Proceedings of the 3rd Jerusalem Conference*, pages 515–522, North-Holland Publishing Company, August 1978. (12)

[Leeman 74] George B. Leeman, Jr., William W. Carter, and Alexander Birman. Some Techniques for Microprogram Validation. In Jack L. Rosenfeld, editor, *Information Processing 74*, pages 76–79, IFIP, North-Holland Publishing Company, August 1974. (12)

[Leeman 75] George B. Leeman, Jr. Some Problems in Certifying Microprograms. *IEEE Transactions on Computers*, C-24(5):545–553, May 1975. (12)

[Levy 84] Beth Levy. Microcode Verification using SDVS. The Method and a Case Study. In *Proceedings of the 17th Annual Microprogramming Workshop*, pages 234–245, 1984. Appeared as a special issue of *Sigmicro Newsletter*, 15(4), December 1984. (13)

[Marcus 84] Leo Marcus, Stephen D. Crocker, and Jaisook R. Landauer. SDVS: A System for Verifying Microcode Correctness. In *Proceedings of the 17th Annual Microprogramming Workshop*, pages 246–255, 1984. (13)

[Maurer 74] W. D. Maurer. Some Correctness Principles for Machine Language Programs and Microprograms. In *Proceedings of the 7th Annual Microprogramming Workshop*, pages 225–234, 1974. (18)

[Maurer 76] W. D. Maurer. Proving the Correctness of a Flight-Director Program for an Airborne Minicomputer. In *Proceedings of the ACM SIGMINI/SIGPLAN Interface Meeting on Program Systems in the Small Processor Environment*, pages 103–108, 1976. Appeared as a special issue of SIGPLAN Notice V11(4), April 1976. (18)

[Maurer 77] W. D. Maurer. An IBM 370 Assembly Language Verifier. In P. A. Willis, editor, *Proceedings of the 16th Annual Technical Symposium on Systems and Software: Operational Reliability and Performance Assurance*, pages 139–146, ACM, June 1977. (18)

[May 87] David May and David Shepherd. *Formal Verification of the IMS T800 Microprocessor.* Technical Report, inmos ltd., 1987. (3,10)

[Moore 88] J. S. Moore. *A Mechanically Verified Language Implementation.* Technical Report 30, Computional Logic Inc., September 1988. (8)

[O'Neill 88] I. M. O'Neill, D. L. Clutterbuck, P. F. Farrow, P. G. Summers, and W. C. Dolman. The Formal Verification of Safety-Critical Assembly Code. In W. D. Ehrenberger, editor, *Proceedings of the IFAC Symposium on Safety of Computer Control Systems 1988 (Safecomp '88) Safety Related Computers in an Expanding Market*, 1988. (18)

[Patterson 78] David A. Patterson. An Approach to Firmware Engineering. In Saki P. Ghosh and Leonard Y. Liu, editors, *AFIPS Conference Proceedings 1978 National Computer Conference*, pages 643–647, Volume 47, AFIPS Press, June 1978. (3,9)

[Patterson 79]	David A. Patterson. STRUM: Structured Microprogram Development System for Correct Firmware. *IEEE Transactions on Computers*, 25(10):974–985, October 1979. (8)
[Patterson 81]	David A. Patterson. An Experiment in High Level Language Microprogramming and Verification. *Communications of the ACM*, 24(10):699–709, October 1981. (8, 9)
[Ramamoorthy 74]	C. V. Ramamoorthy and K. S. Shankar. Automatic Testing for the Correctness and Equivalence of Loopfree Microprograms. *IEEE Transactions on Computers*, C-23(8):768–782, August 1974. (17)
[Richards 86]	Martin Richards. *BSPL A Language for Describing the Behaviour of Synchronous Hardware.* Technical Report 84, University of Cambridge Computer Laboratory, July 1986. (19, 25, 29, 37, 123, 162, 169, 203)
[Roscoe 86]	A. W. Roscoe and C. A. R. Hoare. *The Laws of Occam Programming.* Technical Report PRG-53, Oxford University Computer Laboratory Programming Research Group, 1986. (10)
[Shepherd 88]	David Shepherd. Using Mathematical Logic and Formal Methods to Write Correct Microcode. In *Proceedings of the 20th Annual Workshop on Microprogramming*, 1988. Appeared as a special issue of *Sigmicro Newsletter*, 19(1 and 2), June 1988. (10)
[van-Mierop 78]	Dono van-Mierop, Leo Marcus, and Stephen D. Crocker. Verification of the FTSC Microprogram. In *Proceedings of the 11th Annual Microprogramming Workshop*, page 118, 1978. Appeared as a special issue of *Sigmicro Newsletter*, 9(4), December 1978. (13)
[Wagner 83]	Alan Wagner and Subrata Dasgupta. Axiomatic Proof Rules for a Machine Specific Programming Language. In *Proceedings of the 16th Annual Microprogramming Workshop*, pages 151–158, 1983. (9)
[Wortman 79]	David B. Wortman. On Legality Assertions in Euclid. *IEEE Transactions on Software Engineering*, SE-5(4):359–367, July 1979. (51)

Appendix A

The Hoare Rules

HOARE.1 (Skip)

$$\texttt{|-}\ \{P\}\ \texttt{SKIP}\ \{P\}$$

HOARE.2 (Abort)

$$\texttt{|-}\ \{\mathrm{F}\}\ \texttt{ABORT}\ \{\mathrm{T}\}$$

HOARE.3 (Sequencing)

$$\frac{\texttt{|-}\ \{P\}\ C_1\ \{\mathrm{R}\} \quad \texttt{|-}\ \{R\}\ C_2\ \{Q\}}{\texttt{|-}\ \{P\}\ C_1\ \texttt{;}\ C_2\ \{Q\}}$$

HOARE.4 (Check)

$$\texttt{|-}\ \{P\ \texttt{/\textbackslash}\ \texttt{IS\_ONEP(}e\texttt{)}\}\ \texttt{CHECK}\ e\ \{P\}$$

The predicate IS_ONEP has value T if its word argument consists only of 1 signals and has value F otherwise.

HOARE.5 (Mi)

$$\frac{\texttt{|-}\ \{P\}\ \texttt{CHECK e ;}\ HP\ \{Q\}}{\texttt{|-}\ \{P\}\ \texttt{MI e}\ \{Q\}}$$

where *HP* is the host program.

HOARE.6 (Assignment)

$$\texttt{|-}\ \{Q[e/v]\}\ v\ \texttt{==}\ e\ \{Q\}$$

HOARE.7 (Parallel assignment)

$$\texttt{|-}\ \{Q[e_1/v_1, \ldots e_n/v_n]\}\ \texttt{PAR \$(}v_1\ \texttt{:=}\ e_1 \ldots v_n\ \texttt{:=}\ e_n\ \texttt{\$)}\{Q\}$$

HOARE.8 (Block)

$$\frac{\texttt{|-}\ \{P\ \texttt{/\textbackslash}\ l_1 = U_1\ \texttt{/\textbackslash} \ldots \texttt{/\textbackslash}\ l_n = U_n\ \}\ C\ \{Q[U_1/l_1\ , \ldots U_n/l_n]\}}{\texttt{|-}\ \{P\}\ \texttt{BLOCK}\ C\ \{Q\}}$$

HOARE.9 (Calling modules)

$$\frac{\texttt{|-}\ \{P\}\ c_m\ \{Q\}}{\texttt{|-}\ \{P\}\ \texttt{CALL\_MODULE}\ m\ \{Q\}}$$

where c_m is the body of module m.

HOARE.10 (If)

$$\frac{\begin{array}{l}\texttt{|-}\ \{P\ \texttt{/\textbackslash}\ \texttt{IS\_ONEP(}b_1\texttt{)}\}\ C_1\ \{Q\}\\ \quad .\\ \quad .\\ \texttt{|-}\ \{P\ \texttt{/\textbackslash}\ \texttt{IS\_ONEP(}b_n\texttt{)}\}\ C_n\ \{Q\}\\ \texttt{|-}\ \{P\ \texttt{/\textbackslash}\ \texttt{IS\_ZEROP(}b_1\texttt{, . . ,}b_n\texttt{)}\}\ C_{n+1}\ \{Q\}\end{array}}{\begin{array}{l}\texttt{|-}\ \{P\ \texttt{/\textbackslash}\ \texttt{BINARYP(}b_1\texttt{, . . ,}b_n\texttt{)}\ \texttt{/\textbackslash}\ \texttt{EXCLUSIVEP(}b_1\texttt{, . . ,}b_n\texttt{)}\}\\ \quad \texttt{IF || }b_1\texttt{ => }C_1\texttt{ || . . }b_n\texttt{ => }C_n\texttt{ DEFAULT }C_{n+1}\\ \quad \{Q\}\end{array}}$$

The predicate `IS_ZEROP` has value `T` if its word argument consists only of 0 signals and has value `F` otherwise.

The predicate `BINARYP` has value `T` if its word argument consists only of 1 and 0 signals and has value `F` otherwise.

The predicate `EXCLUSIVEP` has value `T` if its word argument consists only of a single 1 signal where all other signals are 0, and has value `F` otherwise.

HOARE.11 (Case)

$$\frac{\begin{array}{l}\texttt{|-}\ \{P\ \texttt{/\textbackslash}\ e\ \texttt{=}\ b_1\}\ C_1\ \{Q\}\\ \quad .\\ \quad .\\ \texttt{|-}\ \{P\ \texttt{/\textbackslash}\ e\ \texttt{=}\ b_n\}\ C_n\ \{Q\}\\ \texttt{|-}\ \{P\ \texttt{/\textbackslash}\ \texttt{\~{}}\ e\ \texttt{=}\ b_1\ \texttt{/\textbackslash}\ .\ .\ \texttt{\~{}}\ e\ \texttt{=}\ b_n\}\ C_{n+1}\ \{Q\}\end{array}}{\begin{array}{l}\texttt{|-}\ \{P\ \texttt{/\textbackslash}\ \texttt{BINARYP(}e\texttt{)}\}\\ \quad \texttt{CASE }e\texttt{ || }b_1\texttt{ => }C_1\texttt{ || . . }b_n\texttt{ => }C_n\texttt{ DEFAULT }C_{n+1}\\ \quad \{Q\}\end{array}}$$

HOARE.12 (While)

$$\frac{\texttt{|-}\ \{P\ \texttt{/\textbackslash}\ \texttt{IS\_ONEP(}b\texttt{)}\}\ C\ \{P\ \texttt{/\textbackslash}\ \texttt{BINARYP(}b\texttt{)}\}}{\texttt{|-}\ \{P\ \texttt{/\textbackslash}\ \texttt{BINARYP(}b\texttt{)}\}\ \texttt{WHILE}\ b\ \texttt{DO}\ \{P\}\ C\ \{P\ \texttt{/\textbackslash}\ \texttt{IS\_ZEROP(}b\texttt{)}\}}$$

HOARE.13 (Postcondition weakening)

$$\frac{\texttt{|-}\ \{P\}\ C\ \{Q'\}\qquad \texttt{|-}\ Q'\ \texttt{==>}\ Q}{\texttt{|-}\ \{P\}\ C\ \{Q\}}$$

HOARE.14 (Precondition strengthening)

$$\frac{\texttt{|-}\ P\ \texttt{==>}\ P'\qquad \texttt{|-}\ \{P'\}\ C\ \{Q\}}{\texttt{|-}\ \{P\}\ C\ \{Q\}}$$

Appendix B

Freeness Rules

The rules below are designed to be easily integrated into a simple recursive algorithm to determine whether an expression could never return X, Z or U values. They are conservative in nature. In many cases more powerful rules could be developed which catch cases when a particular value could not be returned which is missed by those below. In general those below appear to be sufficient however.

In this and the subsequent appendices, the notation $\mathcal{L}[\![e]\!]$ is used to denote the length of expression e.

B.1 Freeness from U

UFREE.1

A constant expression will not return U values provided it does not contain a U signal

$$\frac{w \texttt{ contains no U signals}}{\texttt{|- FREE\_FROM\_U } w}$$

UFREE.2

Some operators will never return U values.

$$\texttt{|- FREE\_FROM\_U (SW } e\texttt{)}$$
$$\texttt{|- FREE\_FROM\_U (GET(}h;t\texttt{))}$$
$$\texttt{|- FREE\_FROM\_U (Z\_WORD } n\texttt{)}$$

UFREE.3

Some operators will only return U values if the operands do.

$$\frac{\begin{array}{c}\texttt{|- FREE\_FROM\_U (}e\texttt{)}\\ \texttt{|- FREE\_FROM\_U (}f\texttt{)}\end{array}}{\begin{array}{c}\texttt{|- FREE\_FROM\_U (}e\texttt{[}s_1\texttt{, . . ,}s_n\texttt{])}\\ \texttt{|- FREE\_FROM\_U (WD } e\texttt{)}\\ \texttt{|- FREE\_FROM\_U (}e\texttt{, }f\texttt{)}\end{array}}$$

UFREE.4

A number of diadic operators will not return U values, provided their operands return neither Z nor U values.

$$\begin{array}{c}
\texttt{|- FREE\_FROM\_U}\ (e)\\
\texttt{|- FREE\_FROM\_Z}\ (e)\\
\texttt{|- FREE\_FROM\_U}\ (f)\\
\texttt{|- FREE\_FROM\_Z}\ (f)\\
\hline
\texttt{|- FREE\_FROM\_U}\ (e\ \texttt{-}\ f)\\
\texttt{|- FREE\_FROM\_U}\ (e\ \texttt{+}\ f)\\
\texttt{|- FREE\_FROM\_U}\ (e\ \texttt{.=}\ f)\\
\texttt{|- FREE\_FROM\_U}\ (e\ \texttt{.\textasciitilde=}\ f)\\
\texttt{|- FREE\_FROM\_U}\ (e\ \texttt{.<}\ f)\\
\texttt{|- FREE\_FROM\_U}\ (e\ \texttt{.>}\ f)\\
\texttt{|- FREE\_FROM\_U}\ (e\ \texttt{.<=}\ f)\\
\texttt{|- FREE\_FROM\_U}\ (e\ \texttt{.>=}\ f)\\
\texttt{|- FREE\_FROM\_U}\ (e\ \texttt{|}\ f)\\
\texttt{|- FREE\_FROM\_U}\ (e\ \texttt{\&}\ f)\\
\texttt{|- FREE\_FROM\_U}\ (e\ \texttt{EQV}\ f)\\
\texttt{|- FREE\_FROM\_U}\ (e\ \texttt{NEQV}\ f)
\end{array}$$

UFREE.5

A number of monadic operators will not return U values, provided their operand returns neither Z nor U values.

$$\begin{array}{c}
\texttt{|- FREE\_FROM\_U}\ (e)\\
\texttt{|- FREE\_FROM\_Z}\ (e)\\
\hline
\texttt{|- FREE\_FROM\_U}\ (\texttt{-}e)\\
\texttt{|- FREE\_FROM\_U}\ (\texttt{+}\ e)\\
\texttt{|- FREE\_FROM\_U}\ (\texttt{.\textasciitilde}\ e)
\end{array}$$

UFREE.6

The shift operators will not return U values provided that neither operand contains U values, and the second operand does not contain Z values.

$$\begin{array}{c}
\texttt{|- FREE\_FROM\_U}\ (e)\\
\texttt{|- FREE\_FROM\_U}\ (f)\\
\texttt{|- FREE\_FROM\_Z}\ (f)\\
\hline
\texttt{|- FREE\_FROM\_U}\ (e\ \texttt{>>}\ f)\\
\texttt{|- FREE\_FROM\_U}\ (e\ \texttt{<<}\ f)
\end{array}$$

UFREE.7

The subscription operator will not return U values provided its first operand will not and its last operand returns neither U nor Z values. The maximum possible address given by the last operand must also be within the size of the first operand.

$$\begin{array}{c}
\texttt{|- FREE\_FROM\_U}\ (e)\\
\texttt{|- FREE\_FROM\_U}\ (f)\\
\texttt{|- FREE\_FROM\_Z}\ (f)\\
n * 2^{\mathcal{L}[f]} <= \mathcal{L}[\ e\ ]\\
\hline
\texttt{|- FREE\_FROM\_U}\ (e\ \texttt{!}n\ f)
\end{array}$$

UFREE.8

The switch expression will not return U values provided that non of the operands return U or Z values. It should be noted that it would be a syntax error for any of the guard constants to include U, Z or X signals.

$$\frac{\texttt{|- FREE\_FROM\_U}\ (s) \quad \texttt{|- FREE\_FROM\_Z}\ (s) \quad \texttt{|- FREE\_FROM\_U}\ (e_i) \quad \texttt{|- FREE\_FROM\_Z}\ (e_i)}{\texttt{|- FREE\_FROM\_U (SWITCH}\ s\ \texttt{||}\ g_1\ \texttt{=>}\ e_1\ .\ .\ .\ \texttt{DEFAULT}\ e_n\texttt{)}}$$

UFREE.9

A guarded expression with only a single guard and the default case will not return U values provided that the guard cannot return X, Z or U values and neither possible result expression can return U values.

$$\frac{\texttt{|- FREE\_FROM\_U}\ (g) \quad \texttt{|- FREE\_FROM\_Z}\ (g) \quad \texttt{|- FREE\_FROM\_X}\ (g) \quad \texttt{|- FREE\_FROM\_U}\ (e_1) \quad \texttt{|- FREE\_FROM\_U}\ (e_2)}{\texttt{|- FREE\_FROM\_U (GUARDED ||}\ g\ \texttt{=>}\ e_1\ \texttt{DEFAULT}\ e_2\texttt{)}}$$

UFREE.10

The subscription update operator does not return U values provided the first operand does not return X, Z or U signals and the second does not return U values.

$$\frac{\texttt{|- FREE\_FROM\_U}\ (e) \quad \texttt{|- FREE\_FROM\_Z}\ (e) \quad \texttt{|- FREE\_FROM\_X}\ (e) \quad \texttt{|- FREE\_FROM\_U}\ (f)}{\texttt{|- FREE\_FROM\_U}\ (e\ \texttt{!=}\ f)}$$

B.2 Freeness from Z

ZFREE.1

A constant expression will not return Z values provided it does not contain a Z signal

$$\frac{w\ \texttt{contains no Z signals}}{\texttt{|- FREE\_FROM\_Z}\ w}$$

ZFREE.2

The majority of operators will never return Z values.

```
|- FREE_FROM_Z (-e)
|- FREE_FROM_Z (e - f)
|- FREE_FROM_Z (e + f)
|- FREE_FROM_Z (+ e)
|- FREE_FROM_Z (e .= f)
|- FREE_FROM_Z (e .~= f)
|- FREE_FROM_Z (e .< f)
|- FREE_FROM_Z (e .> f)
|- FREE_FROM_Z (e .<= f)
|- FREE_FROM_Z (e .>= f)
|- FREE_FROM_Z (e | f)
|- FREE_FROM_Z (e & f)
|- FREE_FROM_Z (e EQV f)
|- FREE_FROM_Z (e NEQV f)
|- FREE_FROM_Z (.~ e)
|- FREE_FROM_Z (SW e)
|- FREE_FROM_Z (WD e)
```

ZFREE.3

The shift and subscription operators will not return Z values if the first argument is free from Z signals.

```
   |- FREE_FROM_Z (e)
-----------------------
|- FREE_FROM_Z (e !n f)
|- FREE_FROM_Z (e >> f)
|- FREE_FROM_Z (e << f)
```

ZFREE.4

Concatenation and selection will not return Z values provided their operands do not.

```
         |- FREE_FROM_Z (e)
         |- FREE_FROM_Z (f)
-----------------------------------
        |- FREE_FROM_Z (e , f)
|- FREE_FROM_Z (e[s1, . . ,sn] )
```

B.3 Freeness from X

XFREE.1

A constant expression will not return X values provided it does not contain a X signal

```
w contains no X signals
-----------------------
   |- FREE_FROM_X w
```

Appendix C

Rewrite Rules

In the descriptions of rewrite rules below, the following conventions are used:

- *w* represents a constant word,
- *e, f, g* represent arbitrary expressions of the appropriate type,
- k, m, n, r_i, s_i, q_i are natural numbers,
- *0* represents a word consisting of 0 signals of the appropriate length,
- *F* represents a word consisting of 1 signals of the appropriate length,
- *Z* represents a word consisting of Z signals of the appropriate length,
- *EVALUATE* [*e*] gives the constant expression resulting from evaluating a constant expression e,
- *INTVAL w* represents the integer corresponding to the constant, binary word w,
- *ww . . w* is used to represent a single word resulting from the concatenation of w to itself some number of times.

A rewrite rule is given either in the form

$$a \longrightarrow b$$

meaning *a* may be unconditionally rewritten to *b*, or in the form

$$\frac{\begin{array}{c} g_1 \\ \cdot \\ \cdot \\ g_n \end{array}}{a \longrightarrow b}$$

which states that *a* rewrites to *b* provided that g_1 . . g_n are true.

C.1 Expression Rewrite Rules

EXP.1 (WD)

Expressions which cannot return U values are well-defined.

$$\frac{\texttt{FREE\_FROM\_U}\ e}{\texttt{WD}\ e \ \longrightarrow\ \texttt{\#b1}}$$

EXP.2 (And-zero1)

Anding with zero yields zero.

$$\frac{\begin{array}{c}\texttt{FREE\_FROM\_U}\ e\\ \texttt{FREE\_FROM\_Z}\ e\end{array}}{0\ \texttt{\&}\ e \ \longrightarrow\ 0}$$

EXP.3 (And-zero2)

Anding with zero yields zero.

$$\frac{\begin{array}{c}\texttt{FREE\_FROM\_U}\ e\\ \texttt{FREE\_FROM\_Z}\ e\end{array}}{e\ \texttt{\&}\ 0 \ \longrightarrow\ 0}$$

EXP.4 (Or-zero1)

Oring with zero yields the other argument.

$$\frac{\begin{array}{c}\texttt{FREE\_FROM\_U}\ e\\ \texttt{FREE\_FROM\_Z}\ e\end{array}}{0\ \texttt{|}\ e \ \longrightarrow\ e}$$

EXP.5 (Or-zero2)

Oring with zero yields the other argument.

$$\frac{\begin{array}{c}\texttt{FREE\_FROM\_U}\ e\\ \texttt{FREE\_FROM\_Z}\ e\end{array}}{e\ \texttt{|}\ 0 \ \longrightarrow\ e}$$

EXP.6 (Neqv-zero1)

Non-equivalence with zero yields the other argument.

$$\frac{\begin{array}{c}\texttt{FREE\_FROM\_U}\ e\\ \texttt{FREE\_FROM\_Z}\ e\end{array}}{0\ \texttt{NEQV}\ e \ \longrightarrow\ e}$$

EXP.7 (Neqv-zero2)

Non-equivalence with zero yields the other argument.

$$\frac{\begin{array}{c}\texttt{FREE\_FROM\_U}\ e\\ \texttt{FREE\_FROM\_Z}\ e\end{array}}{e\ \texttt{NEQV}\ 0 \ \longrightarrow\ e}$$

EXP.8 (Neqv-ones1)

Non-equivalence with ones yields the negation of the other argument.

$$\begin{array}{c} \texttt{FREE\_FROM\_U}\ e \\ \texttt{FREE\_FROM\_Z}\ e \\ \hline F\ \texttt{NEQV}\ e \quad \longrightarrow \quad .\!\sim\ e \end{array}$$

EXP.9 (Neqv-ones2)

Non-equivalence with ones yields the not of the other argument.

$$\begin{array}{c} \texttt{FREE\_FROM\_U}\ e \\ \texttt{FREE\_FROM\_Z}\ e \\ \hline e\ \texttt{NEQV}\ F \quad \longrightarrow \quad .\!\sim\ e \end{array}$$

EXP.10 (Not-not)

Taking the not of a not expression yields the same result as the argument of the inner not.

$$\begin{array}{c} \texttt{FREE\_FROM\_U}\ e \\ \texttt{FREE\_FROM\_Z}\ e \\ \hline .\!\sim\ (.\!\sim\ e) \quad \longrightarrow \quad e \end{array}$$

EXP.11 (Plus-zero)

Adding zero to an expression leaves it unchanged, unless the addition returns U values.

$$\begin{array}{c} \texttt{FREE\_FROM\_U}\ e \\ \texttt{FREE\_FROM\_Z}\ e \\ \hline 0\ \texttt{+}\ e \quad \longrightarrow \quad e \end{array}$$

EXP.12 (Rsh-const)

Shifting right is the same as concatenating with zero at the most significant end and then selecting the appropriate number of high order signals.

$$\begin{array}{c} \texttt{BINARYP}\ w \\ \texttt{FREE\_FROM\_U}\ e \\ \hline e\ \texttt{<<}\ w \quad \longrightarrow \quad (0,\ e)[k+\mathcal{L}[\ a\ ]\ -1..k] \end{array}$$

EXP.13 (WSW)

Converting a state variable expression to a word and back again, leaves it unchanged.

$$\texttt{WS (SW}\ e\texttt{)} \quad \longrightarrow \quad e$$

EXP.14 (Merge-zword)

Merging with a word of Z signals just returns the old value.

$$\texttt{MERGE(}e\texttt{; Z\_WORD}\ n\texttt{)} \quad \longrightarrow \quad e$$

EXP.15 (Merge-non-z)

Merging with non Z signals, returns the new value.

$$\begin{array}{c} \texttt{FREE\_FROM\_Z}\ f \\ \texttt{FREE\_FROM\_U}\ f \\ \hline \texttt{MERGE(}e\texttt{;}\ f\texttt{)} \quad \longrightarrow \quad \texttt{WS}\ f \end{array}$$

EXP.16 (Subscript)

Subscripting with a known address, is equivalent to selecting the appropriate signals.

$$\frac{\begin{array}{c}\texttt{FREE\_FROM\_U}\ e\\ \texttt{BINARYP}\ w\\ k = \mathit{INTVAL}\ w\ k * n < \mathcal{L}[\ e\ ]\end{array}}{e\ \texttt{!}n\ w\ \longrightarrow\ e\texttt{[}((k+1)*n)-1,.\ \ .,\ (k*n)\texttt{]}}$$

EXP.17 (Subscript-const)

Taking the subscript of a constant which consists of a repeated set of signals of the length subscripted results in a word consisting of that set of signals.

$$\frac{\begin{array}{c}\texttt{FREE\_FROM\_U}\ e\\ \texttt{FREE\_FROM\_Z}\ e\\ \mathcal{L}[\ w\ ]\ = n\\ n * 2^{\mathcal{L}[e]} <= \mathcal{L}[\ ww.\ .w\ ]\end{array}}{ww.\ .w\ \texttt{!}n\ e\ \longrightarrow\ w}$$

EXP.18 (Subscript-concat1)

Subscription when the most significant part of the address is known is the same as subscripting on a smaller subject.

$$\frac{\begin{array}{c}\texttt{BINARYP}\ w\\ \texttt{FREE\_FROM\_U}\ e\\ c = \texttt{INTVAL}\ w\\ (1 + c) * n < \mathcal{L}[\ e\ ]\\ n_1 = 2^{(\mathcal{L}[w]+\mathcal{L}[f])*n}\\ n_2 = 2^{\mathcal{L}[f]*n}\\ k = ((c + 1) * n_2) - 1\\ m = c * n_2\end{array}}{e\ \texttt{!}n\ (w,\ f)\ \longrightarrow\ (e\texttt{[}k,k\text{-}1,\ .\ \ .\ \ ,m\texttt{]})\ \texttt{!}n\ f}$$

EXP.19 (Subscript-concat2)

Subscription when part of the address is known is the same as subscripting on a smaller subject.

$$\frac{\begin{array}{c}\texttt{BINARYP}\ w\\ \texttt{FREE\_FROM\_U}\ e\\ c = \mathit{INTVAL}\ w\\ (1 + c) * n < \mathcal{L}[\ e\ ]\\ n_1 = 2^{(\mathcal{L}[w]+\mathcal{L}[g])*n}\\ n_2 = 2^{\mathcal{L}[g]*n}\\ k = ((c + 1) * n_2) - 1\\ m = c * n_2\end{array}}{e\ \texttt{!}n\ (f,\ w,\ g)\ \longrightarrow\ ((e\ \texttt{!}n_1\ f)\texttt{[}k,k\text{-}1,.\ \ .,m\texttt{]})\ \texttt{!}n\ g}$$

EXP.20 (Subscript-merge-match)

Subscripting into an updated state word at the location updated yields the updated value.

$$\frac{\texttt{FREE\_FROM\_U}\ g}{\texttt{(SW (MERGE(}e\texttt{; }f\ \texttt{!=}\ g\texttt{))) !}n\ f \longrightarrow g}$$

EXP.21 (Select-all)

Selection of all signals from a word, in the right order is just that word.

$$\frac{\mathcal{L}[\ e\ ]\ = n + 1}{e\texttt{[}n\texttt{,}n\texttt{-}1\texttt{,. .,0]} \longrightarrow e}$$

EXP.22 (Select-zword)

Selection of signals from a word of Z signals will be an appropriately sized constant of Z signals.

$$\texttt{(Z\_WORD } n\texttt{)[}s_1\texttt{,. .,}s_n\texttt{]} \longrightarrow \texttt{\#Z. .Z}$$

EXP.23 (Select-select)

Selecting from a selection, is the same as selecting the appropriate signals.

$$\frac{\begin{array}{c}\texttt{FREE\_FROM\_U}\ e\\ q_i \text{ is the } r_i\text{-th from the list } [s_1,.\ .,s_n]\\ \text{counting from the } s_k\text{-th entry as the zeroth}\end{array}}{e\texttt{[}s_1\texttt{,. .,}s_k\texttt{][}r_1\texttt{,. .,}r_m\texttt{]} \longrightarrow e\texttt{[}q_1\texttt{,. .,}q_n\texttt{]}}$$

For example, a[2,3][1] is rewritten to a[2].

EXP.24 (Select-subscript)

Selecting from a subscripted constant is the same as subscripting over a smaller constant with the inaccessible bits stripped out.

$$\frac{\begin{array}{c}\texttt{FREE\_FROM\_U}\ w\\ w' = w \text{ with those bits that cannot be selected stripped out}\end{array}}{\texttt{(}w\ \texttt{!}n\ e\texttt{)[}s_1\texttt{,. .,}s_n\texttt{]} \longrightarrow w'\ \texttt{!}m\ e}$$

If w has the form $w_1\ w_2\ w_3.\ .w_k$ where each of the w_i are length n sections of w, then w' has the form $w_1\texttt{[}s_1\texttt{,. .,}s_m\texttt{]}.\ .w_k\texttt{[}s_1\texttt{,. .,}s_m\texttt{]}$

EXP.25 (Select-concat-low)

Selecting from a concatenation, in which all the selections refer to the low order expression in the concatenation, is the same as selecting the same bits from that expression.

$$\frac{\begin{array}{c}\texttt{FREE\_FROM\_U}\ e\\ \text{for each } i,\ s_i < \mathcal{L}[\ f\ ]\end{array}}{\texttt{(}e\texttt{, }f\texttt{)[}s_1\texttt{,. .,}s_n\texttt{]} \longrightarrow f\texttt{[}s_1\texttt{,. .,}s_n\texttt{]}}$$

EXP.26 (Select-concat-high)

Selecting from a concatenation, in which all the selections refer to the high order expression in the concatenation, is the same as selecting those bits offset by the type of the low order expression, from the high order expression.

$$\frac{\begin{array}{c}\texttt{FREE\_FROM\_U}\ f\\ \text{for each } i,\ s_i > \mathcal{L}[\ f\ ]\ -1\end{array}}{\texttt{(}e\texttt{, }f\texttt{)[}s_1\texttt{,. .,}s_n\texttt{]} \longrightarrow e\texttt{[}s_1-\mathcal{L}[\ f\ ]\texttt{,. .,}s_n-\mathcal{L}[\ f\ ]\ \texttt{]}}$$

EXP.27 (Select-merge-match)

Making a selection from an updated state word at the locations that have been updated gives the updated signals

$$\frac{\begin{array}{c}\texttt{BINARYP}\ w\\ \texttt{FREE\_FROM\_U}\ f\\ k = \mathit{INTVAL}\ w\\ r_1 >= r_2\\ r_1 <= ((k+1) * \mathcal{L}[\,f\,]\,) - 1\\ r_2 >= k * \mathcal{L}[\,f\,]\\ s_1 = r_1 - (k * \mathcal{L}[\,f\,]\,)\\ s_2 = r_2 - (k * \mathcal{L}[\,f\,]\,)\end{array}}{\texttt{(SW MERGE(}e\texttt{;}\ w\ \texttt{!=}\ f\texttt{))[}r_1\texttt{,}\ r_1\text{-}1\texttt{,}\ .\ \ .\ \texttt{,}r_2\texttt{]} \longrightarrow f\texttt{[}s_1\texttt{,}s_1-1\texttt{,}.\ \ .\texttt{,}s_2\texttt{]}}$$

EXP.28 (Select-merge-no-match)

Making a selection from an updated state word at the locations that have not been updated is the same as selecting from the original state word

$$\frac{\begin{array}{c}\texttt{BINARYP}\ w\\ \texttt{FREE\_FROM\_U}\ f\\ k = \mathit{INTVAL}\ w\\ r_1 >= r_2\\ (r_2 > ((k+1) * \mathcal{L}[\,f\,]\,) - 1) \vee (r_2 < k * \mathcal{L}[\,f\,]\,)\end{array}}{\texttt{(SW MERGE(}e\texttt{;}\ w\ \texttt{!=}\ f\texttt{))[}r_1\texttt{,}\ r_1\text{-}1\texttt{,}.\ \ .\texttt{,}r_2\texttt{]} \longrightarrow e\texttt{[}r_1\texttt{,}\ r_1\text{-}1\texttt{,}.\ \ .\texttt{,}r_2\texttt{]}}$$

EXP.29 (Select-switch)

Selections can be moved inside a switch expression, selecting on each possible result.

```
(SWITCH e                            SWITCH e
 || w₁ => e₁                          || w₁ => e₁[s₁,. .,sₘ]
  .                       ⟶            .
  .                                    .
 || wₙ => eₙ                          || wₙ => eₙ[s₁,. .,sₘ]
 DEFAULT eₙ₊₁)[s₁,. .,sₘ]             DEFAULT eₙ₊₁[s₁,. .,sₘ]
```

EXP.30 (Concat-assoc)

Concatenation is associative.

$$(e,\ f),\ g \longrightarrow e,\ (f,\ g)$$

EXP.31 (Concat-concat-const)

Adjacent constants within chains of concatenations can be evaluated.

$$\frac{w_3 = \mathit{EVALUATE}\ [\ w_1,\ w_2\ ]}{w_1,\ (w_2,\ e) \longrightarrow w_3,\ e}$$

EXP.32 (Concat-select)

Concatenating two selections with the same subject, is the same as making all the selections on the subject.

$$e[r_1,.\ \ .,\ r_m],\ e[s_1,.\ \ .,\ s_n] \longrightarrow e[r_1,.\ \ .,\ r_m,\ s_1,.\ \ .,\ s_n]$$

EXP.33 (Switch-constant)

Switching on a constant which is binary, is the same as the appropriate expression.

$$\frac{\begin{array}{c}\texttt{BINARYP}\ w\\ w\ \texttt{=}\ w_i\end{array}}{\begin{array}{l}\texttt{SWITCH}\ w\\ \ \texttt{||}\ w_1\ \texttt{=>}\ e_1\\ \qquad .\\ \qquad .\\ \ \texttt{||}\ w_n\ \texttt{=>}\ e_n\\ \texttt{DEFAULT}\ e_{n+1}\end{array}\quad\longrightarrow\quad e_i}$$

EXP.34 (Switch-default)

Switching on a constant which is binary but matches no guard, is the same as the default expression.

$$\frac{\begin{array}{c}\texttt{BINARYP}\ w\\ \text{for no } i,\ (w\ \texttt{=}\ w_i)\end{array}}{\begin{array}{l}\texttt{SWITCH}\ w\\ \ \texttt{||}\ w_1\ \texttt{=>}\ e_1\\ \qquad .\\ \qquad .\\ \ \texttt{||}\ w_n\ \texttt{=>}\ e_n\\ \texttt{DEFAULT}\ e_{n+1}\end{array}\quad\longrightarrow\quad e_{n+1}}$$

EXP.35 (Switch-identical)

If all result expressions in a switch are identical, and the expression switched on does not contain U or Z signals, then the switch returns that result expression.

$$\frac{\begin{array}{c}\texttt{FREE\_FROM\_Z}\ e\\ \texttt{FREE\_FROM\_U}\ e\end{array}}{\begin{array}{l}\texttt{SWITCH}\ e\\ \ \texttt{||}\ w_1\ \texttt{=>}\ w\\ \qquad .\\ \qquad .\\ \ \texttt{||}\ w_1\ \texttt{=>}\ w\\ \texttt{DEFAULT}\ w\end{array}\quad\longrightarrow\quad w}$$

EXP.36 (Guarded-default)

A guarded expression with only a default clause, returns the value of the default expression.

$$\texttt{GUARDED DEFAULT}\ e \longrightarrow e$$

EXP.37 (Guarded-one)

If a guarded expression has only one guard and it is #b1, then that clauses expression is the result.

$$\texttt{GUARDED || \#b1 =>}\ e_1\ \texttt{DEFAULT}\ e_2 \longrightarrow e_1$$

EXP.38 (Guarded-zero)

Zero guarded clauses may be removed.

```
GUARDED                          GUARDED
 || g1 => e1                      || g1 => e1
        .                                .
 || #b0 => e          ⟶                  .
        .                         || gn => en
 || gn => en                      DEFAULT en+1
 DEFAULT en+1
```

C.2 Condition Rewrite Rules

COND.1 (Not_E_eq_E)

An equality between distinct constant words rewrites to `F`.

$$\frac{w_1 \text{ and } w_2 \text{ are distinct constants}}{w_1 \texttt{ = } w_2 \longrightarrow \texttt{F}}$$

COND.2 (Binaryp_T)

A constant containing only 0 and 1 signals is binary.

$$\frac{w \text{ is a constant containing only 1 and 0 signals.}}{\texttt{BINARYP}(w) \longrightarrow \texttt{T}}$$

COND.3 (Is_onep_T)

IS_ONEP of a constant word consisting only of the signal 1 is true.

$$\frac{w \text{ is a constant consisting only of 1 signals}}{\texttt{IS\_ONEP}(w) \longrightarrow \texttt{T}}$$

COND.4 (Is_zerop_T)

IS_ZEROP of a zero constant is true.

$$\frac{w \text{ is a constant consisting only of 0 signals}}{\texttt{IS\_ZEROP}(w) \longrightarrow \texttt{T}}$$

COND.5 (T_Conj_P)

Conjunction with `T` is equivalent to the other expression.

$$\texttt{T /\textbackslash}\ P \longrightarrow P$$

COND.6 (P_Conj_T)

Conjunction with `T` is equivalent to the other expression.

$$P\ \texttt{/\textbackslash\ T} \longrightarrow P$$

COND.7 (F_Conj_P)

Conjunction with `F` is equivalent to `F`.

$$\texttt{F /\textbackslash}\ P \longrightarrow \texttt{F}$$

COND.8 (P_Conj_F)

Conjunction with `F` is equivalent to `F`.

$$P \text{ /\textbackslash } \mathtt{F} \longrightarrow \mathtt{F}$$

COND.9 (T_Disj_P)

Disjunction with `T` is equivalent to `T`.

$$\mathtt{T} \text{ \textbackslash/ } P \longrightarrow \mathtt{T}$$

COND.10 (P_Disj_T)

Disjunction with `T` is equivalent to `T`.

$$P \text{ \textbackslash/ } \mathtt{T} \longrightarrow \mathtt{T}$$

COND.11 (F_Disj_P)

Disjunction with `F` is equivalent to the other condition.

$$\mathtt{F} \text{ \textbackslash/ } P \longrightarrow P$$

COND.12 (P_Disj_F)

Disjunction with `F` is equivalent to the other condition.

$$P \text{ \textbackslash/ } \mathtt{F} \longrightarrow P$$

COND.13 (E_eq_E)

Equality between two identical expressions is `T`

$$e = e \longrightarrow \mathtt{T}$$

COND.14 (Is_onep_wd)

Expressions which cannot contain U signals are well-defined

$$\frac{\mathtt{|-\ FREE\_FROM\_U}\ e}{\mathtt{IS\_ONEP(WD}\ e\mathtt{)} \longrightarrow \mathtt{T}}$$

C.3 Command Rewrite Rules

COM.1 (Check-one)

A check command with argument `#b1` does nothing

$$\mathtt{CHECK\ \#b1} \longrightarrow \mathtt{SKIP}$$

COM.2 (Par-eq)

Assignments within a parallel assignment which assign the value of a variable to itself may be removed

$$\begin{array}{lcl}
\mathtt{PAR} & & \mathtt{PAR} \\
\mathtt{\$(} & & \mathtt{\$(} \\
\quad v_1 \mathtt{\ :=\ } e_1 & & \quad v_1 \mathtt{\ :=\ } e_1 \\
 & & \\
\quad v \mathtt{\ :=\ } v & \longrightarrow & \\
 & & \quad v_n \mathtt{\ :=\ } e_n \\
\quad v_n \mathtt{\ :=\ } e_n & & \mathtt{\$)} \\
\mathtt{\$)} & &
\end{array}$$

Appendix D

Gordon's Computer's Source Host Program

```
INPUT
        button   : 1
        knob     : 2
        switches : 16



OUTPUT
        pcdisp   : 13
        accdisp  : 16
        ready    : 1
        idle     : 1



STATE_VARIABLES
        arg      : 16
        ir       : 16
        buf      : 16
        mar      : 13
        pc       : 13
        acc      : 16
        mem      : 13 -> 16
        mpc      : 5
```

```
HOST_PROGRAM

rom ==
// rsw
// |wmar
// ||
// || memcntl
// || ||
// || || wpc
// || || |rpc
// || || ||wacc
// || || |||racc
// || || ||||wir
// || || |||||rir
// || || ||||||
// || || |||||| warg
// || || |||||| |
// || || |||||| | alucntl
// || || |||||| | ||
// || || |||||| | || rbuf
// || || |||||| | || |ready
// || || |||||| | || ||idle
// || || |||||| | || |||                      location
// || || |||||| | || ||| aaddr                | source
// || || |||||| | || ||| ||||| baddr          | |    destination   aaddr
// || || |||||| | || ||| ||||| ||||| test     | |    |    function | baddr
// || || |||||| | || ||| ||||| ||||| |||      | |    |    |        |  |
(#b00_00_000000_0_00_000_00000_00000_000, // 31               jmp  0
 #b00_00_000000_0_00_000_00000_00000_000, // 30               jmp  0
 #b00_00_000000_0_00_000_00000_00000_000, // 29               jmp  0
 #b00_00_000000_0_00_000_00000_00000_000, // 28               jmp  0
 #b00_00_000000_0_00_000_00000_00000_000, // 27               jmp  0
 #b00_00_000000_0_00_000_00000_00000_000, // 26               jmp  0
 #b00_10_000100_0_00_000_10001_00000_000, // 25 racc wmem     jmp 17
 #b00_01_001000_0_00_000_10001_00000_000, // 24 rmem wacc     jmp 17
 #b00_01_000000_0_11_000_10101_00000_000, // 23 rmem wbuf sub jmp 21
 #b01_00_000001_0_00_000_10111_00000_000, // 22 rir  wmar     jmp 23
 #b00_00_001000_0_00_100_10001_00000_000, // 21 rbuf wacc     jmp 17
 #b00_01_000000_0_10_000_10101_00000_000, // 20 rmem wbuf add jmp 21
 #b01_00_000001_0_00_000_10100_00000_000, // 19 rir  wmar     jmp 20
 #b00_00_100000_0_00_100_00101_00000_000, // 18 rbuf wpc      jmp  5
 #b00_00_010000_0_01_000_10010_00000_000, // 17 rpc  wbuf inc jmp 18
 #b01_00_000001_0_00_000_11001_00000_000, // 16 rir  wmar     jmp 25
// rsw
// |wmar
// ||
// || memcntl
// || ||
// || || wpc
// || || |rpc
// || || ||wacc
// || || |||racc
// || || ||||wir
// || || |||||rir
// || || ||||||
// || || |||||| warg
// || || |||||| |
// || || |||||| | alucntl
// || || |||||| | ||
// || || |||||| | || rbuf
// || || |||||| | || |ready
// || || |||||| | || ||idle
// || || |||||| | || |||                      location
// || || |||||| | || ||| aaddr                | source
// || || |||||| | || ||| ||||| baddr          | |    destination   aaddr
// || || |||||| | || ||| ||||| ||||| test     | |    |    function | baddr
// || || |||||| | || ||| ||||| ||||| |||      | |    |    |        |  |
 #b01_00_000001_0_00_000_11000_00000_000, // 15 rir  wmar     jmp 24
 #b00_00_000100_1_00_000_10110_00000_000, // 14 racc warg     jmp 22
 #b00_00_000100_1_00_000_10011_00000_000, // 13 racc warg     jmp 19
 #b00_00_000000_0_00_000_10001_01011_010, // 12               jze 17 11
 #b00_00_100001_0_00_000_00101_00000_000, // 11 rir  wpc      jmp  5
 #b00_00_000000_0_00_000_00000_00000_000, // 10               jmp  0
 #b00_00_000000_0_00_000_01010_00000_100, //  9               jop 10
 #b00_01_000010_0_00_000_01001_00000_000, //  8 rmem wir      jmp  9
 #b00_10_000100_0_00_000_00000_00000_000, //  7 racc wmem     jmp  0
 #b01_00_010000_0_00_000_01000_00000_000, //  6 rpc  wmar     jmp  8
 #b00_00_000000_0_00_010_00110_00000_001, //  5 ready         jbut 6  0
 #b01_00_010000_0_00_000_00111_00000_000, //  4 rpc  wmar     jmp  7
 #b10_00_001000_0_00_000_00000_00000_000, //  3 rsw  wacc     jmp  0
 #b10_00_100000_0_00_000_00000_00000_000, //  2 rsw  wpc      jmp  0
 #b00_00_000000_0_00_000_00010_00000_011, //  1               knob 1
 #b00_00_000000_0_00_011_00000_00001_001  //  0 ready idle    jbut 0  1
)
```

```
m       == rom !29 mpc


rsw     == m[28]
wmar    == m[27]
memcntl == m[26..25]
wpc     == m[24]
rpc     == m[23]
wacc    == m[22]
racc    == m[21]
wir     == m[20]
rir     == m[19]
warg    == m[18]
alucntl == m[17..16]
rbuf    == m[15]
ready   == m[14]
idle    == m[13]
aaddr   == m[12..8]
baddr   == m[7..3]
test    == m[2..0]



accdisp == acc
pcdisp  == pc



bus     == GUARDED
            || rsw     .= 1    => switches
            || rpc     .= 1    => #b000,pc
            || racc    .= 1    => acc
            || rir     .= 1    => #b000,ir[12..0]
            || rbuf    .= 1    => buf
            || memcntl .= 1    => mem !16 mar
```

```
mem := SWITCH  memcntl || 2        => mar != bus
acc := SWITCH  wacc    || 1        => bus
ir  := SWITCH  wir     || 1        => bus
arg := SWITCH  warg    || 1        => bus
mar := SWITCH  wmar    || 1        => bus
pc  := SWITCH  wpc     || 1        => bus
```

```
buf := SWITCH alucntl
        || 0        =>       bus
        || 1        => 1   + bus
        || 2        => arg + bus
        || 3        => arg - bus
```

```
mpc := SWITCH test
        || 0        => aaddr
        || 1        => (SWITCH button
                          || 1 => baddr
                          DEFAULT aaddr)
        || 2        => (SWITCH acc
                          || 0 => baddr
                          DEFAULT aaddr)
        || 3        => aaddr + knob
        || 4        => aaddr + ir[15..13]
```

Appendix E

Gordon's Computer's Skeleton

```
TIME
        time
        Start_Time


GHOST
        Button         : 1
        Knob           : 2
        Switches       : 16

        Op             : 3
        Idle           : 1
        Idle_at_Start  : 1

        Mem            : 13->16
        Acc            : 16
        Pc             :  13


INPUT
        button         : 1
        knob           : 2
        switches       : 16


OUTPUT
        pcdisp         : 13
        accdisp        : 16
        ready          : 1
        idle           : 1
```

```
STATE_VARIABLES
        arg         :16
        ir          :16
        buf         :16
        mar         :13
        mpc         :5
        pc          :13
        acc         :16
        mem         :13->16


SKELETON
{
 GHOSTS_SET(mpc; mem; pc; acc; button; knob; switches; time;
            Idle; Start_Time; Button; Knob; Switches; Mem; Acc; Pc;
            Op; Idle_at_Start)  /\
 LOOP_INVARIANT (Idle; mpc; mem; pc; acc; button; knob; switches)
}
$(
  CASE Idle
   || 1 =>
      $(
        MI((SW mpc) .= 0); // ready idle jbut 0 1
        {
         IDLE_START(Idle; Button; Start_Time; Knob; Switches;
                    Mem; Acc; Pc; Op; Idle_at_Start;
                    mpc; mem; pc; acc; time; idle;
                    ready; button; knob; switches)
        }
        CASE Button
         || 1 =>
            $(
              MI ((SW mpc) .= 1);  // knob 1
              {
               IDLE_PRE_CALL(Idle; Button; Mem; Acc; Pc; Knob;
                             Switches; Start_Time; Idle_at_Start;
                             mpc; time; mem; pc; acc; button;
                             knob; switches; ready; idle)
              }
              CASE GET (knob; INC Start_Time)
               || 0 => CALL_MODULE Load_pc
               || 1 => CALL_MODULE Load_acc
               || 2 => CALL_MODULE Load_mem
               DEFAULT CALL_MODULE Run

            $)
         DEFAULT CALL_MODULE Idle_to_Idle
      $)
```

```
    DEFAULT
     $(
       MI ((SW mpc) .= 5); // ready jbut 6 0
       {
        RUN_START(Idle; Op; Mem; Acc; Pc; Button; Start_Time;
                  Idle_at_Start; mpc; mem; pc; acc; button; knob; switches;
                  time; idle; ready)
       }
       CASE Button
        || 1 => CALL_MODULE Run_to_Idle
        DEFAULT
         $(
           MI((SW mpc) .= 6); // rpc wmar jmp 8
           MI((SW mpc) .= 8); // rmem wir jmp 9
           MI((SW mpc) .= 9); // jop 10
           {
            RUN_PRE_CALL(Idle; Button; Mem; Pc; Idle_at_Start; Start_Time;
                         Op; mpc; ir; mar; mem; pc; acc; time;
                         button; knob; switches; idle; ready)
           }
           CASE Op
            || 0 => CALL_MODULE Halt
            || 1 => CALL_MODULE Jmp
            || 2 => CALL_MODULE Jze
            || 3 => CALL_MODULE Add
            || 4 => CALL_MODULE Sub
            || 5 => CALL_MODULE Lda
            || 6 => CALL_MODULE Sta
            DEFAULT CALL_MODULE Skp
         $)
     $)
$)
{
 CYCLE_END(Idle_at_Start;Button;Op;mem;Mem;pc;Pc;acc;Acc;Idle;Knob;
           Switches;Start_Time;time;switches;knob;idle;ready;mpc)  /\
 LOOP_INVARIANT (Idle; mpc; mem; pc; acc; button; knob; switches)
}
```

Appendix F

The Orion Source Host Program

```
// Derived from the BSPL specification by MR.
INPUT

int            : 1       // interrupt request
inrdy          : 1       // byte waiting to be read from DP
outrdy         : 1       // DP ready to receive a byte
dp_inbyte      : 8       // byte from DP
set_csh_parity : 1       // cause a cache parity error
set_tb_parity  : 1       // cause a translation buffer parity error
set_mem_parity : 1       // cause a memory parity error

OUTPUT

inint          : 1       // byte accepted signal
outint         : 1       // byte ready on dp_outbyte
dp_outbyte     : 8       // byte to DP

STATE_VARIABLES

umem        : 15->64     // micro program memory
useg        : 3          // micro program segment register
umar        : 12         // micro program address register
map_mem     : 13->15     // entry point store
instr       : 64         // micro instruction register

ir_h        : 4          // instruction set number
ir_a        : 1          // alternative map
ir_l        : 8          // instruction byte

reg         : 4->32      // 2901 ram registers
q           : 32         // 2901 q register
alt_set     : 1          // force alt SIN set on next cycle
c_save_bit  : 1          // carry save bit
qr_save_bit : 1          // Q R save bit
```

```
div_ff       : 1          // add/sub flag (for division)
scff         : 1          // sign compare flip flop
ab_to_zb     : 1          // disable add (for multiplication)

last_cc      : 1          // last condition code


// 2910 micro program controller registers

counter      : 12         // 2910 register/counter
pc           : 12         // 2910 micro program counter
sp           : 3          // 2910 stack pointer
stack        : 48         // 2910 four words of stack
tos          : 12         // 2910 top of five word stack

// cache

csh_h        : 4          // cache high
csh_a        : 1          // cache alternative page
csh_l        : 9          // cache offset
csh_mem      : 14->32     // cache memory
csh_out      : 32         // cache output register
csh_parity   : 1          // cache parity fault

// translation buffer

var          : 32         // virtual address register
tb_mem       : 11->28     // region tables
mm_mem       : 2->4       // memory mode table
raw_rflt     : 1          // uncorrected read fault
raw_wflt     : 1          // uncorrected write fault
tb_parity    : 1          // translation buffer parity fault

// main memory interface

mem_op       : 3          // latch for m-instr mfunc field
out_reg      : 32         // bus output register
in_reg       : 32         // bus input register
mar          : 26         // main memory address register
mem          : 26->32     // main memory
mem_parity   : 1          // main memory parity fault
prev_lrd     : 1          // previous cycle was read on L mem bank
prev_rrd     : 1          // previous cycle was read on R mem bank
prev_lwr     : 1          // L mem bank has been written to since (R)ADDR
prev_rwr     : 1          // R mem bank has been written to since (R)ADDR
```

```
HOST_PROGRAM


map_mem := Z_WORD 122880
umem    := Z_WORD 2097152


instr          := umem !64 (useg, umar)


a              == instr[3..0]       // 2901 A addr
b              == instr[7..4]       // 2901 B addr
alu_dest       == instr[10..8]      // alu dest
alu_fun        == instr[13..11]     // alu function
alu_source     == instr[16..14]     // alu source
cin            == .~ instr[17]      // alu carry in (inverted)
shifter_mask   == instr[25..23,20]  // shifter mask bits
shifter_n      == instr[19..18]     // shifter count
sin            == instr[22..21]     // 2901 ram and q carry options
speed          == instr[27..26]     // speed
br_addr        == instr[39..28]     // 12 bit microcode address
opcode         == instr[43..40]     // 2910 sequencer function
paritybit      == instr[44]         // parity of microinstruction
cc             == instr[49..46]     // condition code
cc_negate      == instr[45]         // condition code negate
cwr            == instr[50]         // write to cache
ca_ir_sfunc    == instr[55..51]     // ca, ir or sfunc operation
lvar           == .~ instr[56]      // load virtual address register
lbr            == .~ instr[58]      // load bus output register
d              == instr[57,60,59]   // D bus source
mfunc          == instr[63..61]     // memory operation next cycle


br_addr_ext == SWITCH br_addr[11]
               || 1    => #x_FFFF_F, br_addr  // sign extended
               DEFAULT    #x_0000_0, br_addr  //   branch address
```

```
//                       ca
//                       ||| ir
//                       ||| ||| sfunc
//                       ||| ||| ||||| alt_set
//                       ||| ||| ||||| | muldiv                * alt_set
//                       ||| ||| ||||| | |                     |
ca_ir_sfunc_bits == ( #b_000_000_11111_0_0,  // 31                CLRPERR
                      #b_000_000_11110_0_0,  // 30                TBWR
                      #b_000_000_11101_0_0,  // 29                WRMM
                      #b_000_000_11100_0_0,  // 28                ININT
                      #b_000_000_11011_1_0,  // 27              * QSAVE
                      #b_000_000_11010_1_0,  // 26              * RSAVE
                      #b_000_000_11001_1_0,  // 25              * ASIN
                      #b_000_000_11000_1_1,  // 24              * USDIV
                      #b_000_000_10111_0_0,  // 23                OUTINT
                      #b_000_000_10110_0_0,  // 22
                      #b_000_000_10101_0_0,  // 21                CSAVE
                      #b_000_000_10100_0_1,  // 20                TCDIVF
                      #b_000_000_10011_1_1,  // 19              * USDIVF
                      #b_000_000_10010_1_1,  // 18              * DIVL
                      #b_000_000_10001_1_1,  // 17              * TCDIV
                      #b_000_000_10000_1_1,  // 16              * MUL
                      #b_000_101_00000_0_0,  // 15        ALDIR
                      #b_101_000_00000_0_0,  // 14 HLDCA
                      #b_001_000_00000_0_0,  // 13 ALDCA
                      #b_011_010_00000_0_0,  // 12 DECCA  PLDIR
                      #b_011_001_00000_0_0,  // 11 DECCA  LDIR
                      #b_011_000_00000_0_0,  // 10 DECCA
                      #b_000_100_00000_0_0,  //  9        HLDIR
                      #b_010_010_00000_0_0,  //  8 INCCA  PLDIR
                      #b_010_001_00000_0_0,  //  7 INCCA  LDIR
                      #b_010_000_00000_0_0,  //  6 INCCA
                      #b_100_011_00000_0_0,  //  5 LDCA   FETCH
                      #b_100_000_00000_0_0,  //  4 LDCA
                      #b_000_011_00000_0_0,  //  3        FETCH
                      #b_000_010_00000_0_0,  //  2        PLDIR
                      #b_000_001_00000_0_0,  //  1        LDIR
                      #b_000_000_00000_0_0   //  0
                    ) !13 ca_ir_sfunc

ca      == ca_ir_sfunc_bits[12..10]
ir      == ca_ir_sfunc_bits[9..7]
sfunc   == ca_ir_sfunc_bits[6..2]

alt_set := ca_ir_sfunc_bits[1]   // use alternate sin set on next cycle

muldiv  == ca_ir_sfunc_bits[0]   // MUL TCDIV DIVL USDIVF TCDIVF or USDIV
```

```
// cache memory -  form cache address from registers

csh_addr  == csh_h, csh_a, csh_l

cairst    == #b_00,                  // bits 31..30
             csh_addr,               //      29..16
             inrdy, outrdy,          //      15..14
             #b_0,                   //      13
             ir_h, ir_a, ir_l        //      12.. 0


// address translation buffer and protection logic

region == var[31..30]
page   == var[25..10]
offset == var[9..0]

tb_entry  == region, page[8..0]
mem_mode  == mm_mem !4 region
tb_word   == tb_mem !28 tb_entry

tag           == tb_word[27..21]
protection    == tb_word[20..16]
physical_page == tb_word[15..0]

var_reg   == region,          // bits 31..30
             mem_mode,        //      29..26
             page,            //      25..10
             offset           //       9.. 0

tb_out    == protection,      // bits 31..27
             mem_mode[2],     //      26
             physical_page,   //      25..10
             offset           //       9.. 0


raw_rflt  := tb_parity |
             // ok if:    accessed, valid, not bounds
             protection[4,1,0] .~= #b_110 |
             mem_mode[2] |
             tag .~= page[15..9]

raw_wflt  := tb_parity |
             // ok if:    accessed, modified, not RO, valid, not bounds
             protection .~= #b_11010 |
             mem_mode[2] |
             tag .~= page[15..9]
```

```
wrmm      == sfunc.=29                                      // WRMM

tbwr      == sfunc.=30                                      // TBWR

rflt == SWITCH wrmm & lvar | tbwr
         || 1     => #bX
         DEFAULT     raw_rflt



wflt == SWITCH wrmm & lvar | tbwr
         || 1     => #bX
         DEFAULT     raw_wflt






// If ab_to_zb=1 then the alu source is changed as follows:
//               AB to ZB,  AQ to ZQ,  ZA to DQ  and  DA to DZ

alu_source_mod == SWITCH ab_to_zb
                   || 1 =>  alu_source & #b_101  // change AB to ZB etc
                   DEFAULT  alu_source


//              r_from_a              \
//              |r_from_d              + alu r operand
//              ||r_from_0            /
//              |||
//              ||| s_from_a          \
//              ||| |s_from_b          \ alu s operand
//              ||| ||s_from_q         /
//              ||| |||s_from_0       /
//              ||| ||||
srcebits == ( #b010_1000,    //  7   DA   (not the standard 2901 order)
              #b001_1000,    //  6   ZA
              #b010_0001,    //  5   DZ
              #b010_0010,    //  4   DQ
              #b100_0100,    //  3   AB
              #b100_0010,    //  2   AQ
              #b001_0100,    //  1   ZB
              #b001_0010     //  0   ZQ
            ) !7 alu_source_mod
```

```
r_from_a  == srcebits[6]
r_from_d  == srcebits[5]
r_from_0  == srcebits[4]
s_from_a  == srcebits[3]
s_from_b  == srcebits[2]
s_from_q  == srcebits[1]
s_from_0  == srcebits[0]

reg_a     == reg !32 a
reg_b     == reg !32 b




// If div_ff is set the the alu function changes as follows:
//    ADD to SUBR   SUBS to OR   AND to NOTRS   EXOR to EXNOR
alu_fun_mod == SWITCH  div_ff
                 || 1    =>  alu_fun & #b_110  // change ADD to SUBR etc
                 DEFAULT     alu_fun


//             alu_not_r            \ operand complement bits
//             |alu_not_s           /
//             ||
//             || alu_add           \
//             || |alu_or            \ alu main function
//             || ||alu_and          /
//             || |||alu_eqv        /
//             || ||||
opbits == ( #b10_0001, // 7 EXOR   ~R EQV S         R NEQV S
            #b00_0001, // 6 EXNOR   R EQV S         R EQV S
            #b00_0010, // 5 AND     R  &  S         R & S
            #b10_0010, // 4 NOTRS  ~R  &  S        ~R & S
            #b01_1000, // 3 SUB     R  + ~S + c_in  R - S - 1 + c_in
            #b00_0100, // 2 OR      R  |  S         R | S
            #b00_1000, // 1 ADD     R  +  S + c_in  R + S + c_in
            #b10_1000  // 0 SUBR   ~R  +  S + c_in  S - R - 1 + c_in
          ) !6 alu_fun_mod

alu_not_r == opbits[5]
alu_not_s == opbits[4]
alu_add   == opbits[3]
alu_or    == opbits[2]
alu_and   == opbits[1]
alu_eqv   == opbits[0]
```

```
c_in      == cin | c_save_bit | div_ff



//              reg_load             \
//              |reg_left             + write to reg (with shift)
//              ||reg_right          /
//              |||
//              ||| q_load           \
//              ||| |q_left           + set q (with shift)
//              ||| ||q_right        /
//              ||| |||
//              ||| ||| y_from_a     \  y from either a or f
//              ||| ||| |
destbits == ( #b010_000_0,  // 7 RAMU   reg_left
              #b010_010_0,  // 6 RAMQU  reg_left  q_left
              #b001_000_0,  // 5 RAMD   reg_right
              #b001_001_0,  // 4 RAMQD  reg_right q_right
              #b100_000_0,  // 3 RAMF   reg_load
              #b100_000_1,  // 2 RAMA   reg_load            y_from_a
              #b000_000_0,  // 1 NOP
              #b000_100_0   // 0 QREG             q_load
            ) !7 alu_dest

reg_load  == destbits[6]
reg_left  == destbits[5]
reg_right == destbits[4]
q_load    == destbits[3]
q_left    == destbits[2]
q_right   == destbits[1]
y_from_a  == destbits[0]



s == GUARDED
      || s_from_a  =>  reg_a
      || s_from_b  => reg_b
      || s_from_q  => q
      || s_from_0  => 0

sval == SWITCH alu_not_s
         || 1 => .~s
         DEFAULT   s
```

```
FEEDBACK 2
$(
// The D Bus
dbus == SWITCH d
        || 0  => ay          // ALU   alu result
        || 1  => in_reg      // BUS   memory bus input register
        || 2  => br_addr_ext // BR    sign extended branch field
        || 3  => csh_out     // CSH   cache
        || 4  => tb_out      // TB    translation buffer
        || 5  => cairst      // CAIR  CA, IR and i/o status bits
        || 6  => var_reg     // VAR   virtual address register

// The Shift and Mask Unit

d_shifted == SWITCH shifter_n
             || 3       => dbus
             || 2       => (dbus,dbus)>>#x18
             || 1       => (dbus,dbus)>>#x10
             DEFAULT       (dbus,dbus)>>#x08

byte_3 == SWITCH shifter_mask[0]
          || 1  =>  #x00
          DEFAULT   d_shifted !8 #b11

byte_2 == SWITCH shifter_mask[1]
          || 1  =>  #x00
          DEFAULT   d_shifted !8 #b10

byte_1 == SWITCH shifter_mask[2]
          || 1  =>  #x00
          DEFAULT   d_shifted !8 #b01

byte_0 == SWITCH shifter_mask[3]
          || 1  =>  #x00
          DEFAULT     d_shifted !8 #b00


alu_data  == (byte_3 ,byte_2, byte_1, byte_0)

r == GUARDED
      || r_from_a  => reg_a
      || r_from_d  => alu_data   // data from the shift and mask unit
      || r_from_0  => 0

rval == SWITCH alu_not_r
        || 1 => .~ r
        DEFAULT    r
f == GUARDED
        || alu_add    =>  (rval + sval + c_in)[31..0]
        || alu_or     =>   rval | sval
        || alu_and    =>   rval & sval
        || alu_eqv    =>   rval EQV sval

ay == GUARDED
        || y_from_a  => reg_a
        DEFAULT         f
$)


c31 == GUARDED
        || alu_add  =>   rval[30..0] + sval[30..0] + c_in .> #x7FFF_FFFF


c_out == GUARDED
          || alu_add   =>  (rval + sval + c_in)[32]
          || alu_or    =>  c_in | f .~= #xFFFF_FFFF
          || alu_and   =>  c_in | f .~= 0
          || alu_eqv   =>  #bX           // set but not useful


ovr == GUARDED
        || alu_add   =>  c31 .~= c_out
        || alu_or    =>  c_out
        || alu_and   =>  c_out
        || alu_eqv   =>  #bX           // set but not useful


c_save_bit := SWITCH sfunc=21
                || 1     =>  c_out  // CSAVE
                DEFAULT      0


qr_save_bit :=
     GUARDED
      || sfunc.=26 & ( alu_dest.=4 | alu_dest.=5 ) => f[0]
                                               // RSAVE and (RAMQD or RAMD)
      || sfunc.=27 & alu_dest=4                         => q[0]
                                               // QSAVE and RAMQD
        DEFAULT                                           #b_X


csh_parity :=
    SWITCH sfunc.=31
        || 1    => #b0                                  // CLRPERR
        DEFAULT    (GUARDED || set_csh_parity => #b1)
```

```
tb_parity :=
    SWITCH sfunc.=31
        || 1    => #b0   // CLRPERR
        DEFAULT    (GUARDED || set_tb_parity => #b1)


mem_parity :=
    SWITCH sfunc.=31
        || 1    => #b0   // CLRPERR
        DEFAULT    (GUARDED || set_mem_parity => #b1)



// The specification of the special functions for multiplication and
// division has not yet been fully checked.

us_div_bit   == scff NEQV div_ff NEQV c_out

tc_div_bit   == scff NEQV f[31]

ab_to_zb := SWITCH muldiv
             || 1 => (SWITCH sfunc
                       || 16 =>  .~q[0]       // MUL
                       || 17 =>   #b0        // TCDIV
                       || 18 =>   q[0]       // DIVL
                       || 19 =>   #b0       // USDIVF
                       || 20 =>   #b0       // TCDIVF
                       || 24 =>   #b0)        // USDIV
              DEFAULT  #b0


scff := SWITCH muldiv
         || 1 => (SWITCH sfunc
                   || 19 => #b1    // USDIVF
                   || 20 => .~f[31] // TCDIVF
                   || 24 => f[31])        // USDIV
         DEFAULT   #b0

div_ff := SWITCH muldiv
           || 1 => (SWITCH sfunc
                     || 16 => #b0         // MUL
                     || 17 => tc_div_bit // TCDIV
                     || 18 => #b0         // DIVL
                     || 19 => #b1         // USDIVF
                     || 20 => #b0         // TCDIVF
                     || 24 => us_div_bit) // USDIV
           DEFAULT   #b0
```

```
asin == alt_set, sin

cs    == f[31] NEQV ovr


r_lsb ==  SWITCH alu_dest
                                          // RAMU
          || 7 =>    (SWITCH asin
                       || 0 => #b_0                        // ZERO
                       || 1 =>  #b_1                        // ONE
                       || 2 =>  f[31]                       // ROT
                       || 3 =>  #b_0                        // ARI
                       || 4 =>  #b_0                        // US
                       || 5 =>  #b_1                        // DROT
                       || 6 =>  f[31]                       // QRSAVE
                       || 7 =>  #b_0)                       // TC

                                          // RAMQU
          || 6 =>    (SWITCH asin
                       || 0 => q[31]          // ZERO
                       || 1 => q[31]          // ONE
                       || 2 => f[31]          // ROT
                       || 3 => q[31]          // ARI
                       || 4 => q[31]          // US
                       || 5 => q[31]          // DROT
                       || 6 => f[31]          // QRSAVE
                       || 7 => q[31])         // TC


q_lsb == SWITCH alu_dest
                                          // RAMQU
          || 6 =>    (SWITCH asin
                       || 0 => #b_0           // ZERO
                       || 1 => #b_1           // ONE
                       || 2 =>  q[31]          // ROT
                       || 3 =>  #b_0           // ARI
                       || 4 =>  us_div_bit     // US
                       || 5 =>  f[31]          // DROT
                       || 6 =>  #b_0           // QRSAVE
                       || 7 =>  tc_div_bit)    // TC
```

```
r_msb == SWITCH alu_dest
                                    // RAMD
          || 5 =>    (SWITCH asin
                        || 0 => #b_0                              // ZERO
                        || 1 => #b_1                              // ONE
                        || 2 => f[0]                              // ROT
                        || 3 => f[31]                             // ARI
                        || 4 => c_out                             // US
                        || 5 => q[0]                              // DROT
                        || 6 => qr_save_bit                       // QRSAVE
                        || 7 => cs)                                // TC

          || 4 =>    (SWITCH asin
                                       // RAMQD
                        || 0 =>  #b_0     // ZERO
                        || 1 =>  #b_1     // ONE
                        || 2 =>  f[0]     // ROT
                        || 3 =>  f[31]    // ARI
                        || 4 =>  c_out    // US
                        || 5 =>  q[0]     // DROT
                        || 6 =>  qr_save_bit     // QRSAVE
                        || 7 =>  cs)              // TC

q_msb == SWITCH alu_dest
          || 4 =>    (SWITCH asin
                                       // RAMQD
                        || 0 => f[0]     // ZERO
                        || 1 => f[0]     // ONE
                        || 2 => q[0]     // ROT
                        || 3 => f[0]     // ARI
                        || 4 => f[0]     // US
                        || 5 => f[0]     // DROT
                        || 6 => f[0]     // QRSAVE
                        || 7 => f[0])    // TC

reg == GUARDED
        || reg_load   => b != f
        || reg_left   => b != (f, r_lsb)
        || reg_right  => b != (r_msb, f) >> 1

q == GUARDED
      || q_load    => f
      || q_left    => (q, q_lsb)
      || q_right   => (q_msb, q) >> 1

inint     == sfunc.=28
outint    == sfunc.=23
```

```
// instruction register operations
ir_a := SWITCH ir
        || 1     => #b0                          // LDIR
        || 2     => #b0                          // PLDIR
        || 3     => #b0                          // FETCH
        || 5     => #b1                          // ALDIR

ir_l := GUARDED
        || inint & inrdy    =>          dp_inbyte
// a three tick input timing constraint should be given here

        || ir=1     =>    ay !8 #b00                    // LDIR
        || ir=2     =>    ay !8 #b01                    // PLDIR
        || ir=3     =>    #x_01                         // FETCH
        || ir=5     =>    ay !8 #b00                    // ALDIR

ir_h == SWITCH ir
        || 4     => ay[3..0]                     // HLDIR

entry     == map_mem !15 (ir_h, ir_a, ir_l)

// a two tick output timing constraint should be given here
dp_outbyte == GUARDED
             || outint & outrdy =>  ir_l



// cache memory

csh_h := SWITCH ca
        || 5     => ay[3..0]                      // HLDIR

csh_l := SWITCH ca
        || 1     => ay[8..0]                      // ALDCA
        || 2     => csh_l + 1                     // INCCA
        || 3     => csh_l - 1                     // DECCA
        || 4     => ay[8..0]                      // LDCA

csh_a := SWITCH ca
        || 4     => #b0                          // LDCA
        || 1     => #b1                          // ALDCA

csh_mem := GUARDED
          || cwr => csh_addr != dbus                      // CWR

csh_out := SWITCH cwr
          || 1     => dbus
          DEFAULT    csh_mem !32 csh_addr
```

```
// address translation operations

var := GUARDED
        || lvar => ay


tb_mem :=
    GUARDED
     || tbwr =>
          tb_entry !=  page[15..9],  // tag bits
                       dbus[31..27], // protection bits A, M, RO, V and B
                       dbus[25..10]  // physical page number


mm_mem := GUARDED
           || wrmm => ay[31..30] !=  ay[29..26]


// main memory access

//             wflt and rflt                wflt and rflt
//             |                            ||||
//             | wflt only                  ||||  wflt only
//             | |                          ||||  ||||
//             | | rflt only                ||||  ||||  rflt only
//             | | |                        ||||  ||||  ||||
//             | | | no fault               ||||  ||||  ||||  no fault
//             | | | |                      ||||  ||||  ||||  ||||
mem_op := ( #o_4_4_4_7,     // 7   RWR   -> LOCK  LOCK  RWR   RWR
            #o_4_6_4_6,     // 6   RRD   -> LOCK  RRD   LOCK  RRD
            #o_5_5_5_5,     // 5   RADDR -> RADDR RADDR RADDR RADDR
            #o_4_4_4_4,     // 4   LOCK  -> LOCK  LOCK  LOCK  LOCK
            #o_4_4_3_3,     // 3   LWR,  -> LOCK  LOCK  LWR   LWR
            #o_4_2_4_2,     // 2   LRD,  -> LOCK  LRD   LOCK  LRD
            #o_1_1_1_1,     // 1   ADDR  -> ADDR  ADDR  ADDR  ADDR
            #o_0_0_0_0      // 0   IDLE  -> IDLE  IDLE  IDLE  IDLE
          ) !3 (mfunc, raw_wflt, raw_rflt)
```

```
//                    corrected_mem_op (odd)
//                    |
//                    | corrected_mem_op (even)
//                    | |
//                    | |   mem_op
//                    | |   |             odd      even
//                    | |   |             |||      |||
corrected_mem_op == ( #o_7_7, // 7  RWR     -> RWR     RWR
                      #o_6_6, // 6  RRD     -> RRD     RRD
                      #o_5_5, // 5  RADDR   -> RADDR   RADDR
                      #o_4_4, // 4  LOCK    -> LOCK    LOCK
                      #o_7_3, // 3  LWR     -> RWR     LWR
                      #o_6_2, // 2  LRD     -> RRD     LRD
                      #o_1_1, // 1  ADDR    -> ADDR    ADDR
                      #o_0_0  // 0  IDLE    -> IDLE    IDLE
                    ) !3 (mem_op, mar[0])

out_reg := GUARDED || lbr => dbus

mar := SWITCH corrected_mem_op
        || 0  =>  #x_XXXX_XXXX                         // IDLE
        || 1  =>  out_reg                              // ADDR
        || 5  =>  out_reg                              // RADDR


prev_rrd := SWITCH corrected_mem_op
             || 0  => #b_0                             // IDLE
             || 1  => #b_0                             // ADDR
             || 2  => #b_1                             // LRD
             || 3  => #b_1                             // LWR
             || 4  => #b_1                             // LOCK
             || 5  => #b_0                             // RADDR
             || 6  => #b_1                             // RRD
             || 7  => #b_0                             // RWR


prev_lrd := SWITCH corrected_mem_op
             || 0  => #b_0                              // IDLE
             || 1  => #b_0                              // ADDR
             || 2  => #b_1                              // LRD
             || 3  => #b_0                              // LWR
             || 4  => #b_1                              // LOCK
             || 5  => #b_0                              // RADDR
             || 6  => #b_1                              // RRD
             || 7  => #b_1                              // RWR
```

```
prev_lwr := SWITCH corrected_mem_op
            || 1  => #b_0                                       // ADDR
            || 3  => #b_1                                       // LWR
            || 5  => #b_0                                       // RADDR

prev_rwr := SWITCH corrected_mem_op
            || 1  => #b_0                                       // ADDR
            || 5  => #b_0                                       // RADDR
            || 7  => #b_1                                       // RWR

in_reg := SWITCH corrected_mem_op
                                                // LRD
          || 2  => (GUARDED
                     || prev_lrd & prev_lwr=0   =>   mem !32 mar
                     DEFAULT                         #x_XXXX_XXXX)
          || 5 => out_reg                          // RADDR
                                                   // RRD
          || 6 => (GUARDED
                    ||  prev_rrd & prev_rwr=0   =>  mem !32 (mar | 1)
                    DEFAULT                         #x_XXXX_XXXX)

mem := SWITCH corrected_mem_op
       || 3  =>  mar != out_reg                                  // LWR

       || 7  =>  (mar | 1) != out_reg                            // RWR

nmp   == .~ mem_parity
ntbp  == .~ tb_parity
pe    == mem_parity | tb_parity | csh_parity

condition == cc_negate NEQV
            ( nmp,       // 15  NMP     not memory parity
              ntbp,      // 14  NTBP    not translation buffer parity
              ay[27],    // 13  OS      odd short
              int,       // 12  INT     interrupt request
              wflt,      // 11  WFLT    memory write fault
              rflt,      // 10  RFLT    memory read fault
              pe,        //  9  PE      parity error
              ay[26],    //  8  OD      odd byte
              ovr,       //  7  O       overflow
              last_cc,   //  6  LC      last cc
              ay[0],     //  5  ODD     odd result on ay bus
              #b_1,      //  4  T       guaranteed success
              f[31],     //  3  S       alu sign
              c_out,     //  2  C       alu carry out
              cs,        //  1  CS      corrected sign
              f.=0       //  0  Z       alu result zero
            ) !1 cc
```

```
last_cc   := condition


// 2910 micro program controller

ctrl_rom ==
//        pc                        \
//        |br                        \ sources for ma
//        ||cntr (=counter)          / (microprogram address)
//        |||tos (=top of stack)    /
//        ||||
//        |||| cjv                  \  conditional vectored jump
//        |||| |
//        |||| | clear              \
//        |||| | |push               + five word stack operations
//        |||| | ||pop              /
//        |||| | |||
//        |||| | ||| load           \  counter operations
//        |||| | ||| |dec           /
//        |||| | ||| ||
//        |||| | ||| ||      i          \
//        |||| | ||| ||      | condition  + rom address
//        |||| | ||| ||      | | r=0    /
//        |||| | ||| ||      | | |
      ( #b1000_0_001_00,   // F,1,1 TWB     pc   pop
        #b1000_0_001_01,   // F,1,0         pc   pop   dec
        #b0100_0_001_00,   // F,0,1         br   pop
        #b0001_0_000_01,   // F,0,0         tos        dec

        #b1000_0_000_00,   // E,1,1 CONT    pc
        #b1000_0_000_00,   // E,1,0         pc
        #b1000_0_000_00,   // E,0,1         pc
        #b1000_0_000_00,   // E,0,0         pc

        #b1000_0_001_00,   // D,1,1 LOOP    pc   pop
        #b1000_0_001_00,   // D,1,0         pc   pop
        #b0001_0_000_00,   // D,0,1         tos
        #b0001_0_000_00,   // D,0,0         tos

        #b1000_0_000_10,   // C,1,1 LDCT    pc         load
        #b1000_0_000_10,   // C,1,0         pc         load
        #b1000_0_000_10,   // C,0,1         pc         load
        #b1000_0_000_10,   // C,0,0         pc         load

        #b0100_0_001_00,   // B,1,1 CJPP    br   pop
        #b0100_0_001_00,   // B,1,0         br   pop
        #b1000_0_000_00,   // B,0,1         pc
        #b1000_0_000_00,   // B,0,0         pc
```

```
#b0001_0_001_00,   // A,1,1 CRTN    tos  pop
#b0001_0_001_00,   // A,1,0         tos  pop
#b1000_0_000_00,   // A,0,1         pc
#b1000_0_000_00,   // A,0,0         pc

#b1000_0_000_00,   // 9,1,1 RPCT    pc
#b0100_0_000_01,   // 9,1,0         br        dec
#b1000_0_000_00,   // 9,0,1         pc
#b0100_0_000_01,   // 9,0,0         br        dec

#b1000_0_001_00,   // 8,1,1 RFCT    pc   pop
#b0001_0_000_01,   // 8,1,0         tos       dec
#b1000_0_001_00,   // 8,0,1         pc   pop
#b0001_0_000_01,   // 8,0,0         tos       dec

#b0100_0_000_00,   // 7,1,1 JRP     br
#b0100_0_000_00,   // 7,1,0         br
#b0010_0_000_00,   // 7,0,1         cntr
#b0010_0_000_00,   // 7,0,0         cntr

#b0000_1_000_00,   // 6,1,1 CJV     cjv       // not quite
#b0000_1_000_00,   // 6,1,0         cjv       // like the 2910
#b1000_0_000_00,   // 6,0,1         pc        // here
#b1000_0_000_00,   // 6,0,0         pc        // or here

#b0100_0_010_00,   // 5,1,1 JSRP    br   push
#b0100_0_010_00,   // 5,1,0         br   push
#b0010_0_010_00,   // 5,0,1         cntr push
#b0010_0_010_00,   // 5,0,0         cntr push

#b1000_0_010_10,   // 4,1,1 PUSH    pc   push load
#b1000_0_010_10,   // 4,1,0         pc   push load
#b1000_0_010_00,   // 4,0,1         pc   push
#b1000_0_010_00,   // 4,0,0         pc   push

#b0100_0_000_00,   // 3,1,1 CJP     br
#b0100_0_000_00,   // 3,1,0         br
#b1000_0_000_00,   // 3,0,1         pc
#b1000_0_000_00,   // 3,0,0         pc

#b0100_0_000_00,   // 2,1,1 JUMP    br
#b0100_0_000_00,   // 2,1,0         br
#b0100_0_000_00,   // 2,0,1         br
#b0100_0_000_00,   // 2,0,0         br
```

```
        #b0100_0_010_00,   // 1,1,1 CJS     br   push
        #b0100_0_010_00,   // 1,1,0         br   push
        #b1000_0_000_00,   // 1,0,1         pc
        #b1000_0_000_00    // 1,0,0         pc   [illegible]
      )


ctrl_bits  == ctrl_rom !10 (opcode, condition, counter.=0)

ma_from_pc   == ctrl_bits[9]
ma_from_br   == ctrl_bits[8]
ma_from_cntr == ctrl_bits[7]
ma_from_tos  == ctrl_bits[6]

cjv          == ctrl_bits[5]

clear        == ctrl_bits[4]
push         == ctrl_bits[3]
pop          == ctrl_bits[2]

load         == ctrl_bits[1]
dec          == ctrl_bits[0]

ma == GUARDED
        || ma_from_pc   => pc
        || ma_from_br   => br_addr
        || ma_from_cntr => counter
        || ma_from_tos  => tos
        || cjv          => entry[11..0]
        || clear        => 0


useg := GUARDED
         || cjv       =>  entry[14..12]


sp := GUARDED
       || clear => 0
       || push  => (GUARDED ||  sp.<5 => sp + 1)
       || pop   => (GUARDED ||  sp.>0 => sp - 1)
```

```
stack := GUARDED
         || clear                   => #x_XXX_XXX_XXX_XXX
         || push                    => (GUARDED ||  sp.<5 => stack, tos)
         || pop                     => (#b_XXXX_XXXX_XXXX, stack)[47..12]


tos := GUARDED
       || push     =>   pc
       || pop      =>   stack[11..0]


counter := GUARDED
           || load      => br_addr
           || dec       => counter - 1


pc        := ma + 1

umar      := ma
```