Technical Report

UCAM-CL-TR-134
ISSN 1476-2986

Number 134

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Correctness properties of the Viper black model: the second level

Avra Cohn

May 1988

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

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

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Contents

Table of Figures

1 Introduction

This report describes the partially completed correctness proof of the Viper 'block model'. Viper [8,9,10,11,22] is a microprocessor designed by J. Cullyer, C. Pygott and J. Kershaw at the Royal Signals and Radar Establishment in Malvern (henceforth 'RSRE') for use in safety-critical applications such as aviation and nuclear power plant control. To this end, Viper has a particulary simple design about which it is relatively easy to reason using current techniques and models.

The designers, who deserve much credit for the promotion of formal methods, intended from the start that Viper be formally verified. Their idea was to model Viper in a sequence of decreasingly abstract levels, each of which concentrated on some aspect of the design, such as the flow of control, the processing of instructions, and so on. That is, each model would be a specification of the next (less abstract) model, and an implementation of the previous model (if any). The verification effort would then be simplified by being structured according to the sequence of abstraction levels. These models (or levels) of description were characterized by the design team. The first two levels, and part of the third, were written by them in a logical language amenable to reasoning and proof. The top level model was a simple, direct state transformation function – a conditional expression specifying the effect on Viper's registers of processing each class of machine instructions (see [8] and [5]). The lowest level model in the sequence was the circuit structure itself, expressed in the hardware description language ELLA; and there were several levels in between.

To give due credit all around, *all* of the design work for Viper was carried out by the RSRE team. So also was the plan for verifying Viper and for structuring the verification effort, as well as the design, into a sequence of abstraction levels. The RSRE team produced the first two levels of specifications of Viper (and part of the third) in a logical language suitable for verification purposes. Further, J. Cullyer carried out a (more-or-less correct) informal paper-and-pencil correctness proof up to the second abstraction level [9]. Viper was not designed in any part or aspect by the Hardware Verification Group at Cambridge University, despite what one reads or hears in the media. The task of formally and mechanically verifying the Viper design (up to register-transfer level) was sub-contracted by RSRE to members of the Hardware Verification Group at Cambridge University, for a period of about two years (see Acknowledgements), during which time the manufacture of the chip was already underway or indeed completed at various U.K. sites. The mechanical verification task was therefore the full extent of the

Cambridge University participation in the Viper project; and the task was carried out during and after the manufacture of the chip.

Verification was intended to be done in HOL (Higher Order Logic) [2,14,15], a theorem-proving system derived from R. Milner's LCF system (Logic for Computable Functions) [12,21] and based on higher order logic as set out by A. Church [3]. HOL was implemented by M. Gordon at Cambridge University and is currently in use by the Hardware Verification Group at Cambridge University and at various sites throughout the world. 'Verification' was understood by the designers (as by the LCF and HOL communities) to mean complete, formal, logical proof in an explicit and well-understood logic. That is, it means proof in the usual mathematical sense of a sequence of inference steps, and not just simulation or some other non-formal process. Proofs of this sort are constructed interactively in HOL with machine assistance and user-guidance, and not (usually) fully automatically.

A case study in the methodology for the Viper proof was carried out by the author and M. Gordon in 1986; this treated a simple hardware device (in fact, a component of Viper) at several abstraction levels down to and including gate level [4]. The first level of the Viper correctness proof was carried out by the author in 1986-7 [5]. (As mentioned, an earlier informal proof had been done by J. Cullyer.) The formal proof, fully completed, confirmed that the second level of description, the 'major state machine', with certain corrections made, faithfully implemented the top level specification of Viper (again, with certain corrections made). The major state machine was designed to implement each top level state transformation (i.e. the processing of each instruction type) by a sequence of lesser steps, each of which determined the *next* step or else indicated the end of a sequence. The steps were called 'major states'. The major state model concerned itself only with the flow of control in Viper, and not with arithmetic or logical computations. That proof, assisted by the HOL system, consisted of about a million primitive inference steps[1] and took about six person-months to complete. As mentioned, it revealed errors (which were subsequently corrected) in both the major state model and in the top level specification, as presented. These errors, however, did not manifest themselves in the actual Viper chips, so that although the major state model was intended to be a useful link in the chain of abstraction levels, it was of no direct concern to the fabricators of Viper chips.

The block model, described here, *is* of concern to the manufacturers because it directly relates to the circuit design. The block model can be previewed in

[1] On the issue of counting inference steps, see the footnote on page 23.

Figure 1 (page 32); it is a partly pictorial and partly textual (and functional) model consisting of 'blocks' (such as Viper's instruction decoder, its ALU and and its memory), with information passing between blocks, and to/from the outside world, at fixed clock cycles. The functional specifications concern the internal combinational logic of the various blocks, but not their time-behaviour, nor the connection between separate blocks; the pictorial specification fills in the rest of that information. Much as at the major state level, the concept of single instruction types being processed by sequences of steps (major states) is built into the block model; specifically, one of the blocks is a counter representing the major state. In addition, several *minor* states implement each major state in the block model; another block is a counter for *minor* states. Thus there is a yet-finer time-scale at the block level than at the major state level.

The first task in the verification effort is to derive a functional expression of the block model in a formal logic which is suitable for reasoning and proof. This is necessary because it is difficult to reason formally about a schematic diagram indicating the transfer of information to and from its sub-units simultaneously[2].

The second task is to analyze the behaviour of the block model *using* its functional representation. What one ultimately wants to know is (for each instruction type) how *many* minor and major steps must be cycled through before the instruction is fully processed, and what the accumulated effects are on the 'state' of the block model after that number of cycles. These involve extracting from the formal representation (i) the concept of the state of the block machine; (ii) the conditions under which one state leads to another; and (iii) and the assumptions which must be made about initial states and 'normal'[3] behaviour in order to resolve the state transitions. These concepts are implicitly determined once the functional representation is constructed. It must also be shown that the state transition conditions cover all logical possibilities, to ensure that no possible instruction types have been omitted.

The third task is to deduce the results at the *higher* level for each instruction type, under the same conditions that drive the block machine through its major

[2]It is possible to imagine doing this by reasoning about sequences of annotated pictures, but the real problem is not so much the obvious awkwardness of such a method as the lack of a formal semantics of pictures. The 'proof' would not be formal without a clear semantics; that is, we would have to invent and justify a logical calculus whose terms were pictures.

[3]'Normal' behaviour means behaviour which is within the scope of the high level specification. For example, we have to assume in the Viper block proof that no resetting signals come in from the outside world during the course of the block-level processing of instructions; and that the block machine's timeout facility is never invoked. This is because the high level specification itself does not treat these block level contigencies.

and minor steps for that particular instruction type. Then, case by case, the results are compared at the two levels. This requires (i) relating the block level state to the higher level state; and (ii) relating the conditions which drive the block machine to the more abstract conditional choices at the higher level. Whether the third task is achieved, as intended, via the major state machine, or is achieved directly by comparing the results to the top level results, is a technical question discussed in Section 10; in this case the latter seems less complicated.

The first task has been completed and is treated in Section 6: from the pictorial and textual block information provided by RSRE we have derived a fully formal expression of the block model from which we can logically infer the block model's behaviour on the various classes of Viper instuctions. This is done using techniques now standard in hardware verification. This task is perhaps the most interesting part of the analysis, and is an important achievement in itself.

The second task has also been completed (see Sections 7 and 8): the functional expression has been used to describe the state transition conditions of the block machine, and then to logically derive the cumulative behaviour of Viper for each instruction type. There are about 120 sequences of major state transitions to consider, each requiring certain assumptions to be made concerning normal behaviour and initial conditions. Every major state transition of the sequences comprises a sequence of minor state transitions. The large number of major state sequences includes all possible types of computation – additions, shifts, comparisons, and so on – with elaborations such as indexed or looked-up operands, and so on. We have also proved that among this multitude of cases no case has been omitted.

The development of the proof up to this point is not fixed in advance; rather, the major and minor state transition conditions are determined by the various block definitions. These are repeatedly unfolded (under the transition conditions for each instruction type) to produce results not necessarily foreseen. This means that the proof (up to this point) makes rather unsophisticated use of HOL; this is the use indicated by the nature of the problem[4].

The third task has *not* been completed, although some preliminary analysis has been done which indicates that the block results are at least plausible. This is discussed in Section 9. It has proved to be impractical to pursue the third task at present in the absence of (i) better support in HOL for advanced reasoning about intricate bit-string manipulations; and (ii) a better understanding on the

[4]Like LCF, HOL provides a repertoire of tools (and facilities for designing one's own tool-set) for proving conjectures by applying strategies in a subgoaling fashion.

part of the author of microprocessor architecture in general and of Viper's design in particular, in order to relate the differing computations at the high and low levels.

In short, what we have achieved is to have proved some useful *properties* of the block model, without having managed to prove the block model to be an adequate implementation of the top level specification. By describing the cumulative effects on the state of the Viper block model of each instruction class, we have essentially *symbolically executed* the block model (considered as a finite-state machine). This in itself is valuable; the results of the symbolic execution could be used to build a simulator able to jump (in a provably correct way) from result to result without having to derive the minor and major transitions of the block machine again. Of course, in *proving* that each derivation is correct and that all possible cases have been treated, we have done rather more than a symbolic execution; and a symbolic execution in itself was not the aim of the work. Nonetheless, the proof can be considered as a kind of 'quality control' for the symbolic evaluation, assuring that the results are dependable. Furthermore, the functional specification of the block model could be used to prove properties of the Viper block model not manifest in the high level specification; for example, we could infer from our representation the consequences of a reset signal or of a 'bad' initial state.

The proof is very hierarchically structured, as one would expect. First the minor state transitions are analyzed (Section 7); then the major state transitions (Section 8.1), themselves composed of minor state sequences; and finally, the *cumulative* results for major state sequences, which process single instruction types (Section 8.3). The many lemmas required throughout the development form a layered dependence structure. These lemmas are described as they arise. Although it is difficult to find a slice through the proof structure which conveys its interlocking complexity and total bulk, it is clearly impossible (and would be hopelessly boring and repetitious, anyway) to describe the proof as a whole; so as far as possible, a particular instruction type is taken as typical: this is addition with overflow detection.

In the process of performing the proof thus far, a great deal has been learned about managing and properly structuring massive proof efforts. The techniques required did not go beyond anything already standard in HOL circles; and aside from space problems, the proof did not tax the HOL system at all, which is encouraging news about HOL. To date, the block level proof comprises about seven million inference steps and required about one person-year to generate. So far, no

real surprises have emerged about the behaviour of the block machine – but then, surprises would seem likelier to turn up in the third (uncompleted) phase of the project than in the first two. (The preliminary correctness results are described in Sections 8.3 and 9.)

The reader will notice (probably with increasing distress) that this paper is a very straightforward, chronological and (deliberately) detailed account of the proof effort, intended so that anyone wishing to reproduce the proof, or part of it, could use the paper as a reference; and so that dedicated HOL users can examine details. A briefer and more conceptual account is to appear in future [6]. No attempt has been made to explain the design of Viper or the parts and operation of the block machine beyond what is really necessary for describing the formal proof in HOL. Anyone interested in Viper apart from this is referred to the RSRE Viper literature.

Although the paper is meant to stand on its own, it will obviously make more sense in context of the RSRE and HOL literature in general and [5] in particular. A *very* brief introduction to Viper is given in Section 3, and to HOL in Section 4. Though actual HOL tactics and procedures are only suggested in this report, it should be understood that for every theorem and lemma mentioned, a full formal proof in HOL was performed – by the application of a procedure in HOL's meta-language ML. From a research point of view, none of the HOL tactics or methods used are particularly original or interesting.

For an overview of the proof effort, Sections 3, 4, 5, 6.1, 6.2 and the Conclusions section should give some idea of what was proved, and a bit about how.

Appendices containing the HOL versions of the original RSRE definitions of the top level and the block level of Viper are provided for reference. The author claims full responsibility for any typographical or other errors in the translation of the original text (which was in an informally annotated and now-obsolete early prototype of the HOL logic, with some misprints) into the HOL logic. Such errors can be difficult to find. The appendices are supplemented by Figure 1 (page 32), a figure compiled by the author from ten separate RSRE figures. To construct Figure 1, the ten separate figures were connected together according to the coincidence of the names of lines amongst the original figures. They were then topologically rearranged, and completed (for type-correctness) by the addition of an extra block (FSELECT_COMB) as well as six extra internal line names: ram, count, preg, xreg, yreg and areg. The author also claims full responsibility for any errors in the translation into Figure 1 of the original ten figures. All subsequent reasoning about the block

model is based on Figure 1.

Finally, Section 2 contains a discussion of the problems involved in asserting that a chip has been 'verified'. There are great dangers in making such claims if their scope and limitations are misunderstood, as they often are. There is little room for misunderstanding in real-life safety-critical applications. The author strongly urges that Section 2 be read and pondered even if the technical sections are skimmed or skipped.

Several other people have worked on the formal verification of processor designs. Hunt [16] used the Boyer-Moore theorem prover to prove correct the 'FM8501', a microcoded microprocessor that he developed as part of his Ph.D. research. The FM8501 is a machine invented for the purpose of the proof (and not implemented), which therefore has a cleaner specification than Viper; it is roughly as complex as Viper. Joyce [17] has verified Tamarack, another machine invented for proof purposes. Tamarack, though very much simpler than Viper, *has* actually been implemented and fabricated. The proof was also done in HOL. This work is noteworthy because the proof goes all the way down to the transistor level, and is currently being extended up to software levels. An early version of Tamarack was verified by Barrow using his VERIFY system [1], and again by Gordon using LCF_LSM [13]; neither of these two latter systems were based on a clearly delineated logical calculus.

2 The Scope and Limitations of the Proof

When we hear that a chip such as Viper has been 'verified', it is essential to understand exactly what is meant. Several important points sharpen and limit the senses in which a chip (and Viper in particular) can be called verified.

Ideally, one would like to prove that a chip correctly implemented its intended behaviour in all circumstances; we could then claim that the chip's behaviour was predictable and correct. In reality, neither an actual *device* nor an *intention* are objects to which logical reasoning can be applied. The intended behaviour rests in the mind of the architects and is not itself accessible. It can be reported in a formal language, but not with checkable accuracy. At the same time, a material device can only be observed and measured, not verified. It can be described in an abstracted way, and the simplified description verified, but again, there is no way to assure the accuracy of the description. Indeed, the description is bound to be inaccurate in some respects, since it cannot be hoped to mirror an entire

physical situation even at an instant, much less as it evolves through time. In short, verification involves two or more *models* of a device, where the models bear an uncheckable and possibly imperfect relation both to the intended design and to the actual device. This point is not merely a philosophical quibble; errors were found both in the top level specification of Viper *and* in its major state model, none of which was either intended by the designers *or* evident in the manufactured Viper chips (they are discussed in [5]). The errors were fairly minor and quickly repaired, but their presence throws into relief the rather limited sense in which an actual product can be said to have been verified against the architect's design.

That the actual chips appeared not to suffer from the errors found in the models also illustrates the rather academic nature of the research described in [5] and in the present paper. The chips were already in the process of being built by the time the sub-contracted verification work began on the major state proof at Cambridge; and they *had* been built and were being advertised by the time the work described in this report was undertaken. While it is possible in theory that an error in an abstract specification had been reflected in the circuit design given by RSRE to the manufacturers – the abstract specifications were no doubt in the architects' minds while they designed the circuit – it seems more likely that because of the weak links between the abstract specifications, the circuit design process and manufacturers, that problems in the specification would *not* propagate down to chip problems.

At more concrete levels of description, the problem may be further complicated by there not even being provided a description in a formal language. For example, Viper's top level specification and its major state model were both supplied in a logical language; but at the block level, the subject of this report, the description given was partly formally (see the Appendix) and partly pictorially (see Figure 1, page 32). Combining these two parts required some human ingenuity and guesswork (see pages 34 to 35 and 37 to 38). Before verification can be meaningfully applied in such cases, a fully formal description must be produced. Once again, accuracy cannot be checked; the new formal description may be a flawed translation of the pictorial specification, or a flawed combination of picture and text, but there is no sense in which this can be tested. One might thus very well end up proving properties of a formal description bearing an imperfect relation to the intended design. In fact, this *was* a problem in the block level representation of Viper; the author's first attempt at a formal representation of the Viper block diagram involved interchanged line names whose presence was only discovered (rather later in the proof) by an unsystematic inspection. This additional problem of the

accuracy of a representation could appear at the gate level, the transistor level or any other level at which a linguistic description has to be constructed creatively from a pictorial one. It further limits the sense in which a chip can be called verified.

Another point which must be made explicit, given that verification relates a more to a less abstract model, is the level of abstraction and the degree of completeness of the models in question. We say that a device has been verified 'at the major state level' or 'at the register transfer level'; it is not enough to say simply 'verified'. For example, Viper's major state machine has been fully verified with respect to its top level specification, where the major state machine captures the flow of control (implicit at the top level) through the fetch-decode-execute cycle, but does *not* concern itself with any arithmetic or logic computations. The block machine concerns itself with Viper's arithmetic and logical operations, and with the transfer of information between registers and memory; and not with gate connections, transistors, electrical effects, timing problems, and similar areas in which unsuspected errors seem most likely to be found[5]. In addition, the models may be incompletely specified. For example, Viper's highest level model is complete only as regards the processing of instructions, and does not cover resetting or timing-out the machine, or other possible behaviours described at the block level. This restricts any analysis to the high level behaviours alone, again ignoring the more subtle issues. In view of all of this, Viper should not be called verified without reference to the nature of the models used to represent it.

Further, as mentioned, the verification of the block level with respect to the high level has for practical reasons not been completed.

All of these limitations on the use of the word 'verified' are glossed over in advertising claims such as the following [23]:

> "VIPER is the first commercially available micropressor with both a formal specification and a proof that the chip conforms to it."

Such assertions, taken as assurances of the impossibility of design failure in safety-critical applications, could have catastrophic results.

[5]In those areas, enormous amounts of research remain to be done on finding useful, tractable models, even before we begin to verify them.

To summarize,

- A physical *chip* is not an object to which proof meaningfully applies.
- The top level formal specification of Viper (and hence any verification effort) is itself incomplete, covering only the fetch-decode-execute cycle of Viper .
- Viper has been analyzed at best at register-transfer level, i.e. still very abstractly, and not yet at levels at which problems seem likeliest to occur.
- *At* register transfer level, the proof has been only partially completed .

Finally, the correctness of an abstract representation of a chip must be placed in context when we talk about the reliability of physical systems in safety-critical applications. The author claims no expertise in the field of reliability, but it would be irresponsible not to point out the obvious: that this very abstract and limited sense of correctness (the equivalence of a register transfer level model to a functional specification of the fetch-decode-execute cycle) is only one of *many* issues which have to be considered collectively. Aside from possible problems at more concrete levels of description, which have already been mentioned, safety will also depend on factors as yet outside of the world of formal description: these range all the way from issues of social administration and communication, as well as staff training and group behaviour, at one end, to the performance of mechanical and chemical parts, and so on, at the other. One has only to list the mass catastrophes of the last ten years or so to perceive the predominant role played by these extra-logical factors. It is the author's guess (albeit, again, not an expert opinion) that the sort of *a posteriori* abstract design correctness discussed in this paper, though of undoubted importance, forms a relatively small contribution to the overall reliability of real machinery. (This seems so at least at the present stage of research into representation and proof, and with the present weak links between designer, verifier and manufacturer.) That is, using a hardware design verified only down to register-transfer level (and there only partially verified and only in 'normal' situations) as part of the control system in extraordinarily hazardous applications (in which large populations or land masses may be destroyed) does not seem significantly safer than using any other design. The use of the word 'verified' must *under no circumstances* be allowed to confer a false sense of security.

These remarks should be taken as evidence, in our short-sighted times, of the need for further basic research (i.e. the *funding* of further basic research), and

not as pessimism. After all, the HOL system, currently one of very few theorem-provers capable of handling realistic hardware proofs, is directly based on research in pure mathematics and philosophy by Frege, Russell and most directly, Church, many decades ago; and on R. Milner's theorem prover for denotational semantics, a very different application area. The remarks pertain to the current early (but thoroughly optimistic) state of research into the representation and verification of hardware.

3 The Design of Viper

Viper [8,9,10,11,22] is a microprocessor designed at the Royal Signals and Radar Establishment and now commercially available. It is intended for use in safety-critical applications, and has several design features supporting such applications. Viper is hard-wired rather than microcoded, to minimize the number of gates. As mentioned in the introduction, no attempt is made in this report to describe the design of the machine or its unique features; for that, the reader is referred to the Viper literature. In this section we introduce only those aspects of the architecture required for a discussion of the correctness proof of the block level model. Indeed, of all the features at that level, only those concerned with (uninterrupted) instruction processing are described. This is because, as mentioned, the top level specification of Viper provided by the designers itself only covers instruction processing.

3.1 Viper Instructions

Viper has a 32-bit memory. Addresses are 20-bit words, but the memory is addressed by 21-bit words whose the most significant bit distinguishes main from peripheral memory. (Peripheral memory is for input-output operations.) The registers visible to the user are: a 20-bit program counter, a 32-bit accumulator, two 32-bit index registers, a boolean flag (for holding the results of comparisons, etc) and a stopping flag (which normally indicates an error condition).

Instructions are 32-bit words, of which the top twelve bits are the instruction code and the bottom twenty the address. The twelve bits of the instruction encode the following fields:

- **Bit 4**: A 1-bit indication of whether the instruction is a comparison;
- **Bits 0 to 3**: A 4-bit function selector indicating the ALU operation to be computed, according to whether a comparison has been indicated;
- **Bits 10 and 11**: A 2-bit register source selector for the computation, indicating either the program counter, the accumulator or one of the two index registers as the source of one of the operands;
- **Bits 8 and 9**: A two-bit memory source selector for the computation, indicating either literal addressing, content addressing, or addressing offset by the value in one of the two index registers as the method of accessing the operand in memory;
- **Bits 5 to 7**: A 3-bit destination selector to choose a destination for the computation from amongst the accumulator, the two index registers, and the program counter conditionally or unconditionally on the boolean flag.

(Some of these fields can double for other purposes.) The operations are as one would expect: comparisons test numerical less-than and equality between operands; non-comparisons (involving one or two operands) include addition, subtraction, shifts, logical operations, procedure calls, and so on. As mentioned, the example used in this report is addition with overflow detection. In that case, the comparison field holds the value 0 (indicating a non-comparison) and the appropriate function selector value happens to be 5.

3.2 Design Features of Viper

Certain internal registers are used (in the course of executing instructions) which are not accessible from the outside: these include a 32-bit temporary register; a 20-bit register for storing the address field of instructions; and a 12-bit register for storing the instruction code.

Viper accepts certain inputs from the outside world which control its behaviour but are not modelled at the top level. (These are all shown in Figure 1, page 32, prefaced by '`e_`'.) They include a signal for resetting the machine (`e_resetbar`), one for single-stepping it (`e_stepbar`), one for forcing an error (`e_errorbar`), and one for extending read/write cycles (`e_reply`). There are also outputs *to* the world, for viewing certain state values directly – for example, the indications of whether the machine is stopped (`e_stopped`), is fetching an instruction (`e_fetchbar`) or is performing

a computation (e_perform). The boolean flag and the major state can also be read externally, on e_bflag and e_majorstate respectively.

Internally, there is a fixed limit (recorded by a counter, count) to the number of cycles in which the memory can respond, after which an exception occurs (to prevent deadlock due to memory failure)[6]. As mentioned earlier, for establishing the correspondence between the block level and top level models we must assume that certain of these signals are well-behaved; e.g. we assume that the reset signal is false throughout the execution of an instruction, and that at the initial stage in processing an instruction, the timeout-counter is not already at its maximum value.

The Viper chip contains no memory aside from its registers. The block level model, however, includes a simple memory model, since that is the minimum configuration in which it makes sense to talk about executing an instruction (for purposes of verification). The memory model provided is simplified in that it responds in a fixed and minimal number of cycles. The Viper design supports other memory protocols, but these are not modelled by RSRE. This means that the timeout facility (with its counter) is not exercised in the correctness proof.

4 The HOL System

The verification described in this report was carried out in the HOL system ('HOL' standing for 'higher order logic'). In this section we attempt to give just enough information about HOL to make the rest of the report readable; readers curious to know more about the system are referred to [2,14,15].

4.1 An Outline of the HOL System

The HOL system is a version of LCF ('logic for computable functions'). LCF was designed by R. Milner in association with C. Wadsworth, M. Gordon, M. Newey and L. Morris [12,21]. HOL, like LCF, is designed to facilitate the interactive generation of formal proofs. In both systems, a *logic* in which problems can be expressed is interfaced to a *programming language* in which proof procedures and strategies can be encoded. The combination enables deductions in the logic (in the sense of chains of primitive inference steps) to be produced by invocation of programming constructs at a higher level of abstractness. This makes it possible for very long, detailed, complex proofs in the logic to be produced by use of procedures

[6]This is the device verified to gate level as a case study [4].

meaningful to the user of the system – yet without compromising the formality and completeness of the underlying proof. Examples of procedures meaningful to a user might include: unfolding definitions on fixed parameters, normalizing form, case analysis, and rewriting left-to-right using axioms and previously proved theorems.

HOL differs from LCF in the particular logic used. The logic part of HOL is conventional higher-order logic as set out by Church [3]. The version used in the HOL system is oriented towards proofs about hardware only insofar as it provides built-in types, constants and axioms for representing *bit strings*[7]. New types, constants and axioms can be introduced by the user, and organised in logical *theories*, as in LCF. Theorems once proved can be saved in and retieved from theories. Theories themselves are organized into hierarchies in which the types, constants, axioms and theorems of an ancestor theory are accessible from within a descendent theory.

The programming language of HOL is ML (for 'meta-language'), which orginated as the meta-language in the LCF system (though is now well-known in its own right). The *type discipline* of ML ensures that the only way to create *theorems* in the object logic is by performing proofs; theorems have the ML type *thm*, objects of which can only be constructed by the application of inference rules to other theorems or axioms. (Theorems are written with a turnstile, |-, in front of them, and with assumptions to the left of the tunstile.) LCF-style proof is explained more fully in [21].

4.2 The Logic

The HOL system uses the ASCII characters ~, \/ and /\, ==>, !, ?, and \ to represent the logical symbols $\neg$, $\vee$, $\wedge$, $\supset$, $\forall$, $\exists$ and λ respectively.

For the purposes of this paper a *term* of higher-order logic can be one of the following kinds.

- A **variable**;
- A **constant** such as T or F (which represent the truth-values *true* and *false* respectively);
- A **function application** of the form $t_1 t_2$ where the term t_1 is called the *operator* and the term t_2 the *operand*;

[7]They are built in only for convenience; the machinery for defining them 'from scratch' exists.

- An **abstraction** of the form `\x.t` where the variable `x` is called the *bound variable* and the term `t` the *body*;
- A **negation** of the form `~t` where `t` is a term;
- A **conjunction** of the form t_1`/\`t_2 where t_1 and t_2 are terms;
- A **disjunction** of the form t_1`\/`t_2 where t_1 and t_2 are terms;
- An **implication** of the form t_1`==>`t_2 where t_1 and t_2 are terms;
- A **universal quantification** of the form `!x.t` where the variable `x` is the bound variable and the term `t` is the body;
- An **existential quantification** of the form `?x.t` where the variable `x` is the bound variable and the term `t` is the body;
- A **conditional** of the form `t=>`t_1`|`t_2 where `t`, t_1 and t_2 are terms; this has *if-part* `t`, *then-part* t_1 and *else-part* t_2;
- A **local declaration** of the form `let x=`t_1 `in` t_2, where `x` is a variable and t_1 and t_2 are terms; this is provably equivalent to `(\x.`t_2`)`t_1 (see Section 4.4); and
- A **list** of the form `[`t_1`:ty;`t_2`:ty;...;`t_n`:ty]` where `ty` is a *type* (see below). Here t_n is called the zeroth element, t_1 the $(n-1)^{th}$ element or the head[8], and `[`t_2`;...;`t_n`]` the tail.

All terms in HOL have a *type*. The expression `t:ty` means `t` has type `ty`; for example, the expressions `T:bool` and `F:bool` indicate that the truth-values `T` and `F` have type `bool` for boolean, and `3:num` indicates that `3` is a number.

If `ty` is a type then `(ty)list` (also written `ty list`) is the type of lists whose components have type `ty`. If ty_1 and ty_2 are types then ty_1`->`ty_2 is the type of functions whose arguments have type ty_1 and whose results have type ty_2. The cartesian product operator is represented by `#`, so that ty_1`#`ty_2 is the type of pairs whose first components have type ty_1 and second, ty_2.

In this paper, the logical constants `AND`, `OR` and `NOT` used by RSRE are used interchangeably with the corresponding HOL constants.

As indicated earlier, the HOL system provides a number of predefined types and constants for reasoning about hardware. The types include $\texttt{word}_n$, the type of n-bit words and $\texttt{mem}_{n1,n2}$ for memories of n_2-bit words addressed by n_1-bit words. The

[8]The 'reverse' numbering is used to correspond with conventional significance-order of bit-strings

expression `#b`$_{n-1}\cdots$`b`$_0$ (where `b`$_i$ is either `0` or `1`) denotes an n-bit word in which `b`$_0$ is the least significant bit.

The predefined constants used in this paper are shown below.

- `V:bool list->num` converts a list of truth-values to a number;
- `VAL`$_n$`:word`$_n$`->num` converts an n-bit word to a number;
- `BITS`$_n$`:word`$_n$`->bool list` converts an n-bit word to a list of booleans;
- `WORD`$_n$`:num->word`$_n$ converts a number to an n-bit word;
- `FETCH`$_{n1}$`:mem`$_{n1,n2}$`->(word`$_{n1}$`->word`$_{n2}$`)` looks up a word at an address in memory;
- `STORE`$_{n1}$`:word`$_{n1}$`->(word`$_{n2}$`->(mem`$_{n1,n2}$`->mem`$_{n1,n2}$`))` stores a word at an address in memory;
- `EL:num->(bool list->bool)` selects the specified element of a boolean list, where the last (rightmost, in the notation) element of the list is the zeroth;
- `CONS:ty->(ty list->ty list)`, for any type `ty`, constructs a new list `[t;`t_1`;`t_2`;...;`t_n`]` from a list `[`t_1`;`t_2`;...;`t_n`]` and an element `t:ty`, so that `t` is the n^{th} element of the new list;
- `HD:ty list->ty`, for any type `ty`, maps a list `[`t_1`;`t_2`;...;`t_n`]` to `t`$_1$, i.e. to its $(n-1)^{th}$ element;
- `TL:ty list->ty list`, for any type `ty`, maps a list `[`t_1`;`t_2`;...;`t_n`]` to `[`t_2`;...;`t_n`]`;
- `SEG:(num#num)->(bool list->bool list)` returns the specified segment of a *boolean* list between and including the elements whose numbers are given;
- `NOT`$_n$`:word`$_n$`->word`$_n$ inverts the bits of a word;
- `AND`$_n$`:word`$_n$`->word`$_n$`->word`$_n$ conjoins two words bit-wise;
- `OR`$_n$`:word`$_n$`->word`$_n$`->word`$_n$ disjoins two words bit-wise; and
- `ARB` is a constant whose type is instantiable to any type; it denotes an arbitrary unspecified value at each type – this is useful in certain formal expressions.

To make terms more readable, HOL uses certain conventions. One is that a term $t_1 t_2 \cdots t_n$ abbreviates $(\cdots(t_1 t_2)\cdots t_n)$; that is, function application associates to the left. The product operator `#` associates to the right and binds more tightly than the operator `->`. For example, the type of `SEG` could be written simply as `num#num->bool list->bool list`. For another example, the function used in the Viper block model to select the register operand for a computation (page 34) has the type

```
word32#word32#word32#word20#word2->word32
```

which abbreviates

```
(word32#(word32#(word32#(word20#word2))))->word32
```

4.3 The Framework for Expressing the Block Model in HOL

The registers of Viper are represented in HOL using the HOL logic's special types for bit-strings. We choose variable names for the various registers so that:

- `areg:word32` is the accumulator (a-register);
- `xreg:word32` is the first index register (x-register);
- `yreg:word32` is the second index register (y-register);
- `preg:word20` is the program counter (p-register);
- `bflag:bool` is the boolean flag (bflag);
- `ram:mem21_32` is the memory;
- `treg:word32` is the temporary register (t-register);
- `addr:word20` is the address register; and
- `inst:word12` is the instruction register.

The constants (predicates) pertaining to hardware are used to manipulate these; for example, if `FETCH_ABBR(ram,preg)` is an abbreviation we introduce that denotes the 12-bit instruction code part of the word in the memory `ram` pointed to by the program counter `preg`), then

```
~EL 4(FETCH_ABBR(ram,preg))
```

means that the new instruction is not a comparison instruction (see page 15). If in addition

```
WORD4(V(SEG(0,3)(FETCH_ABBR(ram,preg)))) = #0101
```

then (given that the instruction is a non-comparison), the value 5 is defined to indicate the ALU operation: addition with overflow detection. (This is the example computation used throughout the report.)

As mentioned, FETCH_ABBR is a convenient *abbreviation* we introduce in the logic. In terms of the basic bit-manipulation function FETCH21, FETCH_ABBR(ram,preg) stands for

```
EL
4
(BITS12
 (WORD12
  (V
   (SEG
    (20,31)
    (BITS32
     (FETCH21 ram(WORD21(V(CONS F(BITS20 preg))))))))))
```

That is, the new instruction is fetched from the address in the 20-bit program counter (with the twenty-first bit false to indicate main rather than peripheral memory); it is converted into a list of thirty-two boolean values; the upper 12-bit segment (the instruction code) is extracted; its value is considered as a 12-bit string; this is converted to a list of twelve booleans; and the fourth element of the list, the comparison indicator, is selected.

This whole expression (and most subsequent expressions) are pretty-printed for ease of parsing (rather than economy of space!)[9].

Occasionally, the 12-bit instruction field (usually represented by the variable inst) and the other fields are dealt with separately from the 20-bit address. In such cases we write

```
EL 4(BITS12 inst)
WORD4(V(SEG(0,3)(BITS12 inst)))
WORD3(V(SEG(5,7)(BITS12 inst)))
WORD2(V(SEG(8,9)(BITS12 inst)))
WORD2(V(SEG(10,11)(BITS12 inst)))
```

to denote the comparison indicator, the function selection field, the destination selection field, the memory indicator and the register selector, respectively.

[9]Expressions of this sort can sometimes be logically simplified, but this expression is one that actually arises in the analysis, so it is left in the form required.

4.4 Proof in HOL

As an example of HOL features mentioned thus far, consider (in a purely formal way) the simple HOL definition of the block model function which computes the source register for a computation. Given the a-, x-, y- and p- registers as arguments, and the 2-bit source register indicator, the function REGSELECT returns the appropriate register (negated):

```
|- REGSELECT(areg,xreg,yreg,preg,rsel) =
   (let rf = VAL2 rsel in
    ((rf = 0) => NOT32 areg |
     ((rf = 1) => NOT32 xreg |
      ((rf = 2) => NOT32 yreg |
       ((rf = 3) => NOT32(WORD32(VAL20 preg)) | ARB)))))
```

The turnstile indicates either an axiom, a definition or a theorem (here, a definition). Constructs of the logic used include the *let*-construct and the conditional. The constant ARB serves as the final *else*-value of the conditional (which is intended never to be returned, since rsel is a 2-bit word). The HOL bit-string constants used in the definition are: VAL2, NOT32, WORD32 and VAL20.

The basic rules of inference in HOL take the form of ML functions which (roughly speaking) map theorems (and sometimes various parameters as well) to theorems. More elaborate patterns of inference can be constructed from the basic inference rules of the logic by user-designed ML functions. The validity of the compound inferences is preserved by the type system of ML in which *only* inference rules may return theorems as results.

In our example, we can transform the definition of REGSELECT more computationally useful by applying a compound inference pattern (an ML function) which expands the *let*-construct into the equivalent lambda expression, and performs the resulting beta-conversions to deduce a new theorem:

```
|- REGSELECT(areg,xreg,yreg,preg,rsel) =
   ((VAL2 rsel = 0) =>
    NOT32 areg |
    ((VAL2 rsel = 1) =>
     NOT32 xreg |
     ((VAL2 rsel = 2) =>
      NOT32 yreg |
      ((VAL2 rsel = 3) => NOT32(WORD32(VAL20 preg)) | ARB))))
```

Applying the inference pattern actually invokes 520 primitive HOL inference steps (as we have written it) but only requires the user to apply one function to one definition in order to generate the 520-step proof. The pattern applies to *any* definition written using the *let*-construct. This relieves the user of constructing a 520-step chain of substitutions of equals for equals, and so on (which no one would want to bother with) whilst still assuring that an actual proof is done.

(For perspective, the whole block model correctness proof comprises about seven million primitive inferences[10].)

Much of the Viper block proof consists in successive unfoldings of known facts on specific values. For example, the new form of the definition of REGSELECT can be applied to particular values, and the resulting theorem then simplified. We can prove for example that

```
|- REGSELECT(areg,xreg,yreg,preg,#00) = NOT32 areg
```

by applying a compound inference rule which instantiates the definition to a specific value and then simplifies, using axioms and previously established facts about bit-strings and so on. This particular unfolding consists in 89 primitive inferences. (In a case argument based on the various possible values of rsel, the theorem above is used to simplify one of the cases.)

Proofs based on unfolding are an example of a particularly simple and unsophisticated use of HOL: they procede in a *forward* direction, starting without a fixed idea of the result, but applying known procedures. In the block model case, the method allows us to symbolically evaluate the block model on each of the possible instruction types, and so give a complete description of the model's behaviour. In contrast, one often starts with a conjecture – i.e. with an end result in mind – and constructs a proof *backwards* by engaging proof strategies to produce successively simpler subgoals. As a result of this process, a forward proof is then constructed.

5 The Plan for Verification

The original RSRE plan was to establish the correctness of Viper in a series of decreasingly abstract stages.

The top level specification, as mentioned in the introduction, is just a transition function specifying how an abstract state (representing the memory and the visible flags and registers) changes as Viper executes each of the possible instruction types. It thus embodies an operational semantics of the instruction set. The notion of time is implicit in the notion of next state; there are no time variables or clocks.

[10]A much shorter proof could be constructed in this particular case, but the figure of 520 arises from the generality of the rule used, which relies on the full power of left-to-right recursive rewriting to expand *let*-expressions (250 steps), and then looks for possible beta-conversions at any depth in the resulting expression. It seems desirable, since many definitions of varying complexity have to be treated similarly, to trade computational resources for user effort in this way. However, the apparent length of proofs can be greatly and sometimes misleadingly inflated by use of such general procedures.

The basic time unit is quite coarse: namely, the execution of a whole instruction schema.

The next (more concrete) level was called the major state level. At the major state level, an instruction is executed by a sequence of events corresponding to major states of the model and representing the phases of processing of an instruction schema. Each event could affect the visible registers of the high level state or any of several internal registers. (These internal registers were still part of an abstract model of Viper, and did not necessarily correspond to parts of the actual Viper chip.) The next event was determined according to the current event, the visible state, and the internal state; some events were recognizably terminal and some are initial, in the sequences. From all of this we extracted a state transition function representing the major state model, which determined a *graph* of events showing all possible major state sequences of Viper. This led to a correctness statement (proved in HOL) of the following form:

> **If** the major state transition function always gives the next internal state based on the current internal state, and **if** the state at the start time is the initial state, and **if** it requires n major state traversals to return for the first time to the initial state, **then** after n traversals the visible part of the internal state agrees with the new state specified by the top level state transition function.

This was proved by analyzing each path through the graph and comparing the results, under the same conditions which force that path to be taken, with the high level results. There was an explicit notion of time at the major state level in the sequencing and accumulation of effects of the various events; several major state transitions simulated the effect of a single state transition at the high level, i.e. simulated the effect of executing a single instruction. The machine proof was based on a paper and pencil proof by J. Cullyer [9].

The ten (numbered) major states and their associated effects are listed below.

- 0. INDEX: To the temporary register is added the register indicated;
- 1. FETCH: The word indicated by the program counter is got from memory;
- 2. RESET: Various registers are cleared;
- 3. PRECALL: The program counter is stored before a procedure call;
- 4. PERFORM ALU: The appropriate arithmetic-logic operation is done;

- 6. READIO: A peripheral device is read;
- 7. READMEM: The memory is read according to the temporary register;
- 8. STOP: The machine is stopped (because of an error);
- 10. WRITEMEM: The memory is written; and
- 11. WRITEIO: A peripheral device is written.

The theorem expressing the correctness of the major state machine, though not easy to prove, did not say very much; merely that the *flow of control* in the major state machine was correct. There was no computation of values at the major state level – that is, additions, comparisons, shifts, and so on – so the essential correctness of Viper was not really addressed; the proof did not require any analysis of the function representing the arithmetic-logic unit, at either level.

Still less abstract than the major state model is the *block level* model [22], whose proof is described in this paper. At the block stage, we begin to approach the functional units and connectivity of the actual circuit, though still in a very abstract way. The block model is a collection of separate functional units (for example, the arithmetic-logic unit, the decoder, and so on) specified functionally, along with information on their inter-connectedness and timing behaviour, specified pictorially (with labelled schematic diagrams). The combination of the definitions and the pictorial information are given; the block machine's behaviour patterns must be inferred logically from what is supplied. The time scale at the block level is finer again than (but congruent with) the scale at the major state level; each major state transition is implemented by several *minor* state transitions. (There are mechanisms (blocks) to keep track of the major and minor states.) The block model therefore determines another graph, in which each major node consists of several minor event nodes. Before the graph is analyzed, though, it first has to be *derived* from the given information (see Section 6.2). (At the major state level the graph transitions were given explicitly in the definition of the major states.)

At the block level, the behaviour of the arithmetic-logic unit *is* analyzed, so we come closer to the essence of the microprocessor. This means that there is no single major state called 'PERFORM' corresponding to the execution of an arithmetic-logic operation (as at the major state level) but rather a collection of distinct major states for the various arithmetic-logic operations of Viper.

The original plan was to state and prove the equivalence of the high level specification and the major state model on the visible state; and then of the major state

model and the block model on the major state; and finally to make the connection by a some sort of transitivity argument. The first equivalence, as mentioned, has been proved completely, and is described in [5]. The second turned out to be awkward (for technical reasons at least). In fact, by using the methods established in the first proof, a direct comparison of the high level and the block level turns out to be straightforward, and this is what is ultimately wanted anyway. The proof has not been completed up to the equivalence (it would not have been practical with present tools and resources), but only to the point of a complete logical description (analysis) of the graph of the block level. This in itself is very valuable because it can be returned to the designers who can then decide whether the results are as predicated and hoped. Further, it reveals what would be required to complete the proof.

The major state proof is thus not necessary for the block proof, and is not used in this paper, but only mentioned in passing. The block level correctness statement, had it been proved, would have had a similar form to the correctness statement at the major state level, except that it would have had the added complication of assumptions carried along with each case. These assumptions limit the correctness of the block machine to certain normal cases in which, for example, no resetting signals come from the outside world during the time taken to process an instruction. This is discussed in Section 8.3.

6 Representing the Problem

6.1 The High Level

The high level or visible state of Viper

```
(ram,preg,areg,xreg,yreg,bflag,stop)
```

is represented in HOL as an object with the type

```
mem21_32#word20#word32#word32#word32#bool#bool
```

where `ram` represents the memory (of 32-bit words addressed by 21-bit words); `preg` the program counter; `areg`, `xreg` and `yreg` the registers; `bflag` a general-purpose boolean flag; and `stop` the flag indicating an error requiring Viper to stop. The high level specification gives state changes for: illegal instructions (which stop the machine), no-ops, arithmetic-logic operations involving comparisons (whose the results are put in the bflag), other arithmetic-logical operations whose results go into one of the three registers, arithmetic-logical operations whose results go into

the program counter (i.e. jumps), write instructions, and procedure calls. The complete definition is supplied in the Appendix, and explanations can be found in [22]. For purposes of this paper all that is really required is the idea that a visible state (comprising the seven components) is transformed directly by the high level function into a new state upon inspection of the current instruction. The current instruction resides in the memory component of the state, ram, and is pointed to by the program counter component of the state, preg.

An example of a state transformation (which is treated at length in this paper, at the block level) is an addition ALU operation with overflow detection, whose results are placed in one of the three registers *besides* the program counter. In that case, the high level specifies a state in which (i) the program counter is incremented; (ii) the sum of the register source indicated and the memory source is computed and placed in the destination register indicated; and (iii) the value of the stopping flag (i.e. whether the sum overflows) is computed. We abbreviate the memory source as MEM_ABBR(ram,preg) and the register source REGSELECT_ABBR(areg,xreg,yreg,preg,ram)[11].

Results are computed at the high level by a sub-function (called ALU'), the high level function representing the arithmetic-logic unit[12]. The function ALU' maps the function field, the memory selector field, the destination field, the register source, the memory source and the boolean flag to three values: a 32-bit computed value, a possible new value for the boolean flag, and a possible new value for the stopping flag. An example of an ALU non-comparison function is addition with overflow detection. In this case the ALU' function returns as the 32-bit computed value

```
ADD32(REGSELECT_ABBR(areg,xreg,yreg,preg,ram),MEM_ABBR(ram,preg))
```

where the addition function ADD32 (on *any* two 32-bit words r and m) returns

```
TRIM34TO32(WORD34((VAL33(SIGNEXT r)) + (VAL33(SIGNEXT m))))
```

Here, SIGNEXT represents sign-extension:

```
|- SIGNEXT w = WORD33(V(CONS(EL 31(BITS32 w))(BITS32 w)))
```

and TRIM34TO32 simply drops the top two bits of the 34-bit sign-extended sum:

```
|- TRIM34TO32 w = WORD32(V(TL(TL(BITS34 w))))
```

[11]The memory abbreviation requires just the program counter and memory as parameters, as explained above. The register abbreviation needs in addition the four registers from amongst which it selects.

[12]In the appendix, and the RSRE formulation, this function has no prime mark, but we add the prime here to distinguish it from the block level function which has the same general purpose but an entirely different type; it is unfortunate that the same name was originally used.

For the stopping flag, the function ALU' returns:

```
~(EL
  32
  (BITS34
   (WORD34
    ((VAL33(SIGNEXT(REGSELECT_ABBR(areg,xreg,yreg,preg,ram)))) +
     (VAL33(SIGNEXT(MEM_ABBR(ram,preg)))))) =
  EL
  31
  (BITS34
   (WORD34
    ((VAL33(SIGNEXT(REGSELECT_ABBR(areg,xreg,yreg,preg,ram)))) +
     (VAL33(SIGNEXT(MEM_ABBR(ram,preg))))))
```

This represents the overflow condition of the sum; the sum spans bits 0 to 33, and the test concerns bits 31 and 32[13]. When all definitions are unfolded in this example the sum of the register and memory sources works out to be:

```
WORD32
(V
 (TL
  (TL
   (BITS34
    (WORD34
     ((VAL33(SIGNEXT(REGSELECT_ABBR(areg,xreg,yreg,preg,ram)))) +
      (VAL33(SIGNEXT(MEM_ABBR(ram,preg))))))))))
```

That is, the sign-extended sources are added to form a 34-bit word whose top two bits are subsequently dropped. The stopping flag works out to be:

```
~(EL
  32
  (BITS34
   (WORD34
    ((VAL33(SIGNEXT(REGSELECT_ABBR(areg,xreg,yreg,preg,ram)))) +
     (VAL33(SIGNEXT(MEM_ABBR(ram,preg)))))) =
  EL
  31
  (BITS34
   (WORD34
    ((VAL33(SIGNEXT(REGSELECT_ABBR(areg,xreg,yreg,preg,ram)))) +
     (VAL33(SIGNEXT(MEM_ABBR(ram,preg))))))
```

(The boolean flag is not affected.) These expressions, of course, can be further unfolded according to the definition of SIGNEXT.

Another (quite straightforward) ALU operation is a procedure call. The main computed value in that case is the jump address, which is placed in the program counter:

```
WORD20(V(SEG(0,19)(BITS32(MEM_ABBR(ram,preg)))))
```

That is, the address part of the memory source is the address to which to jump. (There is no register source required for a procedure call.)

To illustrate the effect of a comparison type ALU operation on the high level state, we consider first equality test on the source and memory registers. In this case, the result of the comparison is placed in the boolean flag, and it is simply

[13]This is a consequence of the high level definition; *why* the expression represents an overflow is another question.

```
REGSELECT_ABBR(areg,xreg,yreg,preg,ram) = MEM_ABBR(ram,preg)
```

For the less-than test (i.e. whether the value of the register source is less than the value of the memory source), the result of the comparison is

```
EL
32
(BITS34
 (WORD34
  ((VAL33(SIGNEXT(REGSELECT_ABBR(areg,xreg,yreg,preg,ram)))) +
   ((VAL33(SIGNEXT(MEM_ABBR(ram,preg))) = 0) => 0 |
    (VAL33(NOT33(SIGNEXT(MEM_ABBR(ram,preg))))) + 1))))
```

That is, the value of the sign-extended source is added to 0 if the value of the sign-extended memory is 0, and to the incremented value of the negated sign-extended memory value otherwise; and the thirty-second bit of that 34-bit sum is the result. (Given the equality and the less-than tests, all other desired comparisons can be constructed.)

Finally, we consider a simple non-ALU operation: an illegal-instruction stop. Various instuction schemata are not legal – for example, any one which tries to use the spare function fields 13, 14 or 15. Likewise, any one in which the program counter cannot be incremented without overflowing its twenty bits is illegal. For all such illegal instructions the new state specified at the high level will have all other registers unchanged but the stop flag set to true.

In all of these examples, the results are *logically inferred* in HOL from the definitions of the high level function and its sub-functions. This is achieved through a process of gradual unfolding of definitions in each case, using the particular case assumptions to simplify resulting expressions. Exactly the same process of unfolding is carried out at the block level, except that there the block level the representing function is not provided explicitly as at the top level, but has instead to be extracted from a partially pictorial representation before the unfolding process can be undertaken.

The examples of this section are compared to the block level results in Section 9. The first example (addition with overflow detection) is the main example of this report.

It should be stressed that the high level specification of Viper is incomplete as regards the actual Viper chip. The specification applies *only* to the fetch-decode-execute cycle, and ignors capabilities such as resetting, timing-out because of memory faults, pausing, single-stepping, externally forced errors, and so on – all of which are possible in the actual circuit, and all of which are represented in the block model. This means that the scope of any relation between the high level specification and the block model must be limited to the fetch-decode-execute

behaviour of Viper. Thus, because of the minimality of the high level model, any correctness property one states is bound to be rather unrealistic. It would be interesting, though beyond this report, to explore better high level specifications.

6.2 The Block Level

At the block level there are ten functional units, or blocks. These are described and explained in more detail in [22], but in summary their names and general purposes are as follows:

- MEMORY_BLOCK: Models the external memory, ram, in a simplified way;
- EXTERNAL_BLOCK: Interfaces internal lines with the external world;
- DATAREG: Computes the data registers: the visible areg, xreg, yreg or preg (whichever is the destination), as well as three internal registers: the address (addr), instruction (inst) and temporary (treg) registers;
- TIMING_COMB: Generates the timing (i.e. sequencing) values within each major state;
- MINOR: Keeps track of the current minor state;
- MAJOR: Keeps track of the current major state;
- TIMEOUT_BLOCK: Provides a 64-cycle timeout facility[14];
- DECODER: Generates the control values for the whole block based on the instruction code;
- FSELECT_COMB: Extracts the function selector field from the instruction code;
- ALU_COMB: Performs the arithmetic and logical operations to yield a 32-bit data result as well as a 9-bit word encoding the stopping conditions associated with the computation; and
- BANDSTOP_BLOCK: Computes two boolean flags: the bflag of the visible state and a stop flag (not quite the same as in the visible state).

[14]This is not relevant to the present analysis because of the simplified memory model used.

Each of these blocks is given a functional description in [22]; the HOL versions appear in their entirety in the Appendix. Their inter-connectedness is conveyed in Figure 1, below, which is a compendium (laid out slightly differently) of several diagrams from [22], to which the names of the internal lines count, ram, areg, xreg, yreg and preg have been added by the author. Boxes marked 'LATCH' or 'L' are latches, which cause a delay of one clock cycle (one time unit). These memory elements can be thought of as registers. Purely combinatorial blocks and sub-blocks (i.e. those without latches) are shown with the suffix '_COMB'. Patterns of feedback are indicated in the figure. The nine lines to or from the external world appear at the top of the figure (prefaced by 'e_').

To save space, the *types* of the values on the lines are not shown in the diagram, but it can be assumed that they are known. For example, the type of the value on the inst line at any time is word12.

The parts of the Viper block model referred to are explained locally and partially as the need arises, but no attempt is made to present a coherent overall explanation. For that the reader must refer to [22].

6.3 An Example

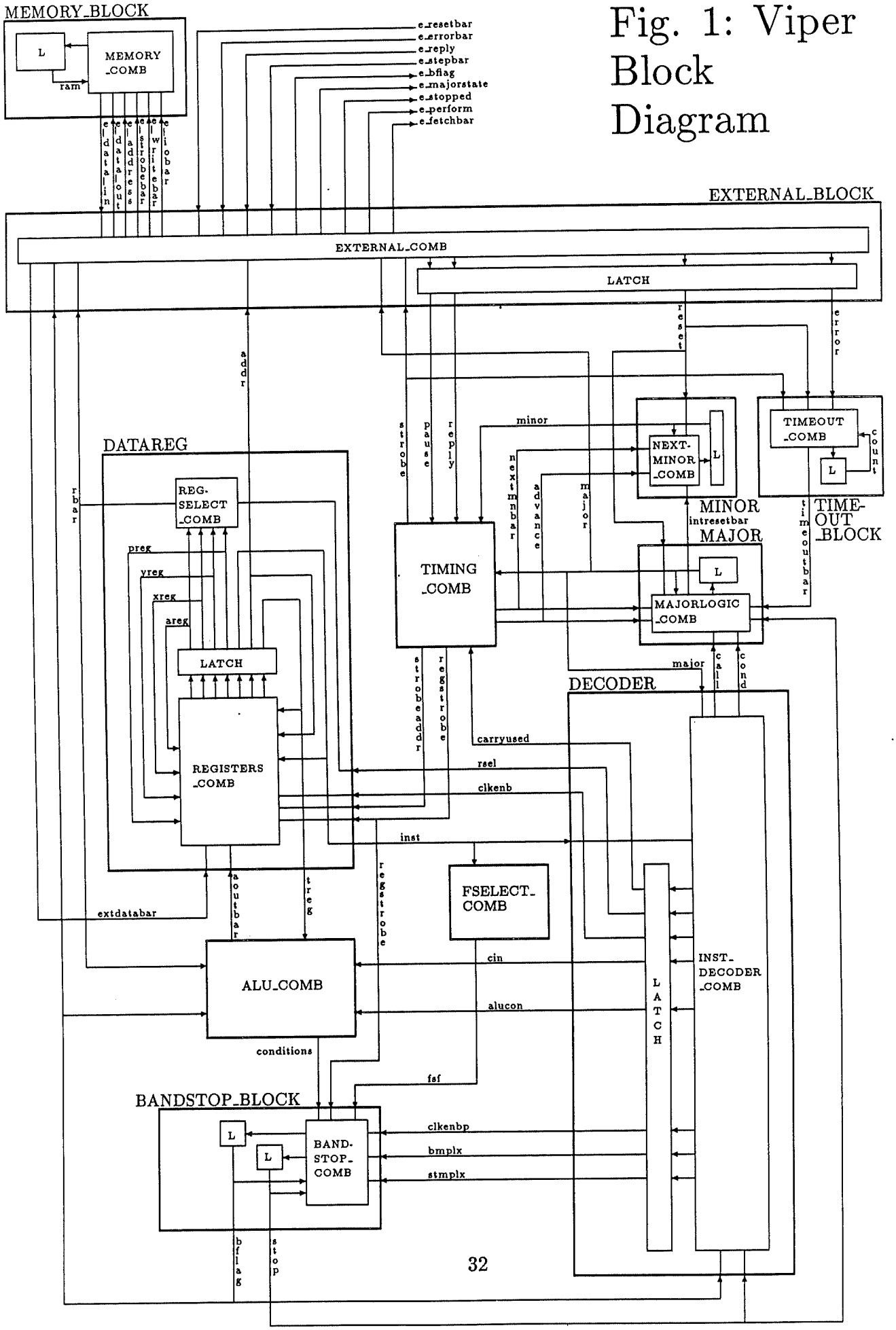
The nature of the block model and its representation in HOL can be illustrated by studying *one* of the blocks. We choose the DATAREG block (which can be found within Figure 1).

By way of motivation, the function of the whole DATAREG block is firstly to remember the four data values of the high level state: areg, xreg, yreg and preg. It is also to remember three internal values: a 32-bit temporary value called treg, a 12-bit value inst for holding the instruction code part of an instruction, and a 20-bit value addr for holding the address part of an instruction. Besides containing these seven registers, the DATAREG block computes new values for them. It outputs *one* of the four visible registers (whichever is indicated by the rsel line from the decoder block).

The new register values are computed on the basis of the old register values, as well as five incoming lines. The informal specification tells us the 'meanings' of these and their relations to the external lines[15]. The five incoming lines are: extdatabar, a 32-bit input from the external interface which, though this is not deducible from the figure, comes from the memory block and represents incoming

[15]This information is represented only in English prose in [22]. None of it is actually necessary for the formal proof.

Fig. 1: Viper Block Diagram



data (e_data_in); aoutbar, the 32-bit data output of the arithmetic-logic unit; clkenb, a line from the decoder controlling when and which registers of DATAREG are to be loaded with new values; and finally, two boolean timing values, regstrobe and strobeaddr, from the timing block, which respectively indicate the stability of an operation, and when the address latch should be loaded.

The main sub-block of the DATAREG block is called REGISTERS (represented by REGISTERS_COMB in Figure 1). That block is described in HOL by a function called REGISTERS which has the following HOL type:

```
word32#(word32#(word7#(bool#(bool#(word32#(word32#(word32#(word20#(word32#(word20#word12)))))))))) ->
word32#(word32#(word32#(word20#(word32#(word20#word12)))))
```

This type, by conventions of binding power (see Section 4) can be abbreviated as

```
word32#word32#word7#bool#bool#word32#word32#word32#word20#word32#word20#word12 ->
word32#word32#word32#word20#word32#word20#word12
```

REGISTERS has the following definition in HOL:

```
|- REGISTERS
   (extdatabar,aoutbar,clkenb,regstrobe,strobeaddr,areg,xreg,yreg,preg,treg,addr,inst) =
   (let extdata = NOT32 extdatabar in
    let aoutbar_ls20 = WORD20(V(SEG(0,19)(BITS32 aoutbar))) in
    let clkenba = NOT(EL 0(BITS7 clkenb)) in
    let clkenbp = NOT(EL 1(BITS7 clkenb)) in
    let clkenbx = NOT(EL 2(BITS7 clkenb)) in
    let clkenby = NOT(EL 3(BITS7 clkenb)) in
    let clkenbinst = NOT(EL 4(BITS7 clkenb)) in
    let clkenbt = NOT(EL 5(BITS7 clkenb)) in
    let tmplxcon = EL 6(BITS7 clkenb) in
    let new_areg = CLOCK_REGA(aoutbar,areg,clkenba,regstrobe) in
    let new_xreg = CLOCK_REGX(aoutbar,xreg,clkenbx,regstrobe) in
    let new_yreg = CLOCK_REGY(aoutbar,yreg,clkenby,regstrobe) in
    let new_preg = CLOCK_REGP(aoutbar_ls20,preg,clkenbp,regstrobe) in
    let new_inst = CLOCK_INST(extdata,inst,clkenbinst,regstrobe) in
    let new_treg = CLOCK_REGT(aoutbar_ls20,extdata,treg,clkenbt,regstrobe,tmplxcon,clkenbinst) in
    let new_addr = CLOCK_ADDR(preg,treg,addr,clkenbinst,strobeaddr) in
    new_areg,new_xreg,new_yreg,new_preg,new_treg,new_addr,new_inst)
```

Here, the various sub-functions will have been previously defined in HOL. For example, the function CLOCK_REGA, which computes the new value of the a-register, has the HOL type

```
word32#word32#bool#bool->word32
```

and is defined to choose between the ALU_COMB block's (negated) data output and the existing value in the a-register on the basis of two booleans values: clkenba, corresponding to the least significant bit of the 7-bit control value clkenb; and regstrobe, from the timing block.

```
|- CLOCK_REGA(aoutbar,areg,clkenba,regstrobe) = ((clkenba /\ regstrobe) => NOT32 aoutbar | areg)
```

Note that the variable names such as new_areg, new_inst and so on, in the definition of REGISTERS, are purely suggestive of their intended meanings. That is, when the *let*-construct is simplified (see Section 4.4), the seventh output of REGISTERS is

```
CLOCK_INST(NOT32 extdatabar,inst,~EL 4(BITS7 clkenb),regstrobe)
```

which gives no indication that the output represents the new value of the inst register (since the function name 'CLOCK_INST' is also mnemonically chosen and does not formally convey a meaning).

The other function of the DATAREG block is REGSELECT. Its HOL definition (as seen earlier) is

```
|- REGSELECT(areg,xreg,yreg,preg,rsel) =
   (let rf = VAL2 rsel in
    ((rf = 0) => NOT32 areg |
     ((rf = 1) => NOT32 xreg |
      ((rf = 2) => NOT32 yreg |
       ((rf = 3) => NOT32(WORD32(VAL20 preg)) | ARB)))))
```

(See Section 4 for the form without *let*'s.) This function takes rsel (the indicator from the decoder for selecting a destination register from amongst the visible data registers) and the four possible destination registers, and returns the appropriate destination register.

So far, three kinds of information appear to be present in Figure 1 that are not present in the functional definition of REGISTERS and REGSELECT alone:

1. The internal latch arrangements of the DATAREG block (i.e. the fact that the seven outputs of REGISTERS_COMB are latched)

2. The relation of the inputs and outputs of the DATAREG block to the inputs and outputs of other blocks (e.g. the fact that the input aoutbar comes from the ALU_COMB block and the output inst goes to the DECODER block).

3. The feedback patterns within the DATAREG block (e.g. the fact that the inst value is fed back from the latch to REGISTERS_COMB) so that the input to REGISTERS_COMB on that line at one time was the output *from* REGISTERS_COMB on that line at the previous time

Figure 1 supplies all of this information pictorially, but is not a formal specification in the sense we need for logical analysis. (In any case, it does not supply the internal definitions of the various blocks.) Regarding the second item, we could guess, for example, from the *form* of the definition of the HOL function defining the ALU_COMB block –

```
|- ALU(rbar,treg,cin,bflag,alucon) =
   (  .
      .
      .
    let aoutbar = ... in
    let conditions = ... in
    aoutbar,conditions)
```

– that the output 'aoutbar' of the ALU_COMB block is identical to the input to DATAREG called by the same name; but of course, that is not a formal specification either, since in both definitions the variable 'aoutbar' is bound, and therefore could in either case could be replaced by a different variable.

Some information present in the RSRE informal descriptions is present neither in Figure 1 *nor* in the formal definitions – for example, as we mentioned above, the connction of the extdatabar line to the e_data_in line. These identities, stated in informal prose in [22], obviously assist one's understanding of Viper, but are not formally a part of its definition, nor indeed are they necessary for the proof. More essential ambiguities are discussed in the next section.

In any case, our goal now is to produce a formal expression representing the whole DATAREG block including its latches, feedback patterns and relations to other blocks.

6.3.1 Representing the Whole Block with the Latch

The function REGISTERS was defined to apply to objects (of various type) representing the values on lines; it took inputs such as 32-bit words and boolean values and returned outputs representing seven registers. To describe the behaviour of the whole DATAREG block, however, we have to introduce the notion of *time* so that we can talk about the latch and its behaviour. To do this, we introduce the notion of a *signal*. Signals are functions from times (represented by natural numbers) to values of appropriate type. Thus, while the block's input extdatabar has type word32, the signal called (by convention) extdatabar_sig is a function with type num->word32. We write extdatabar_sig n to denote the *value* of the function extdatabar_sig at time n.

As a first step towards a *function* representing the DATAREG block, we define a new *relation* called REGISTERS_COMB (the suffix '_COMB', as mentioned earlier, suggesting the purely combinational part of the block). [16]

The relation REGISTERS_COMB holds between the input and output signals corresponding to the input and output values of the function REGISTERS, and it describes the unit's behaviour in time. The output signals of REGISTERS_COMB are given the arbitrary names out1 to out7. The types of these signals can be inferred from the type of REGISTERS: they evaluate respectively to values of type: word32, word32, word32, word20, word32, word20 and word12. The relation REGISTERS_COMB is defined as follows:

[16] From this point on, the development is our own, based on the RSRE definitions and corresponding pictures. The use of relations is a standard hardware verification method.

```
|- REGISTERS_COMB
   (extdatabar_sig,aoutbar_sig,clkenb_sig,regstrobe_sig,strobeaddr_sig,
    areg_sig,xreg_sig,yreg_sig,preg_sig,treg_sig,addr_sig,inst_sig,
    out1_sig,out2_sig,out3_sig,out4_sig,out5_sig,out6_sig,out7_sig) =
   (!n.
     out1_sig n,out2_sig n,out3_sig n,out4_sig n,out5_sig n,out6_sig n,out7_sig n =
     REGISTERS
     (extdatabar_sig n,aoutbar_sig n,clkenb_sig n,regstrobe_sig n,strobeaddr_sig n,
      areg_sig n,xreg_sig n,yreg_sig n,preg_sig n,treg_sig n,addr_sig n,inst_sig n))
```

This means: the relation REGISTERS_COMB holds over its input and output signal arguments if and only if all times, the output values of the function REGISTERS at that time are the result of applying REGISTERS to the input values at that time. (The relation is purely combinational because, as yet, there is no advance in time.) Similarly, for the function REGSELECT, we define a relation

```
|- REGSELECT_COMB(areg_sig,xreg_sig,yreg_sig,preg_sig,rsel_sig,rbar_sig) =
   (!n. rbar_sig n = REGSELECT(areg_sig n,xreg_sig n,yreg_sig n,preg_sig n,rsel_sig n))
```

where the rbar_sig signal returns an object of type word32.

In the block model, latches introduce the only delays. To describe a latch formally in HOL – a 32-bit latch, for example – we introduce a relation LATCH32 between two signals each of type num->word32, and we express the fact that the first signal is latched to the second by asserting that

```
|- !in out. LATCH32(in,out) = (!n. out(n+1) = in n)
```

This means that the output at the next time (one time unit later) is *always* equal to the input at the current time.

Thus far, everything has been based on the functional definitions provided by RSRE. We now turn to the pictures provided. The DATAREG block is shown in Figure 2 below. This figure is just an enlarged extract from Figure 1, to which we have added the internal line names out1 to out7. The labelling of the internals lines, though essential to a correct representation, has to be guessed; the discussion below analyzes this problem in detail.

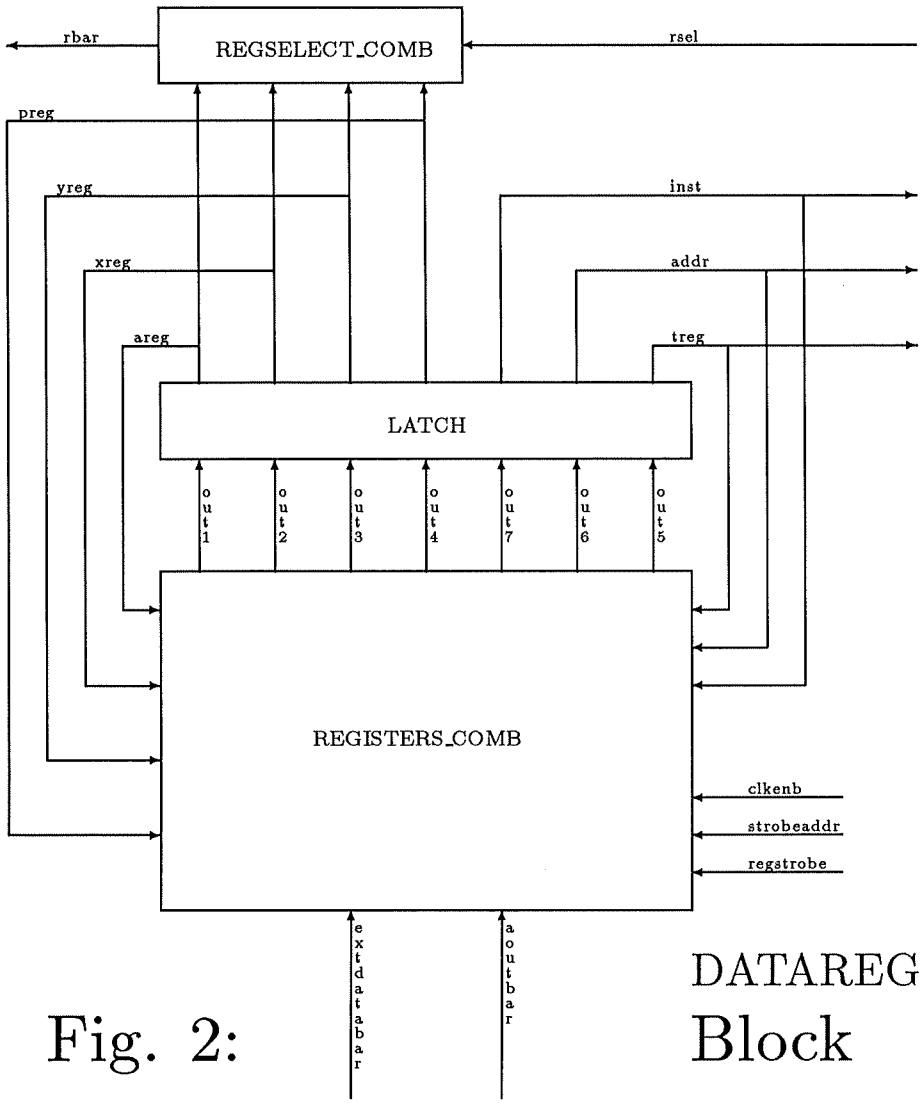


Fig. 2: DATAREG Block

An Aside on Combining Pictures and Text

Considering Figure 2 before the labels `out1` to `out7` are added, it is clear that the seven output signals of `REGISTERS_COMB` are each latched. (The latch block is assumed to be a pictorial abbreviation for seven separate latches. Also, '`out1`' abbreviates '`out1_sig`', and so on.) Four of the output signals of the latch(es) are used only internally (they are fed back to `REGISTERS_COMB`). The other three are fed back *and* are also (named and typed) outputs of the whole `DATAREG` block. These latter three outputs must be uniquely identified with outputs of the textual `REGISTERS_COMB`; since they represent lines to other blocks we need to know how `REGISTERS_COMB` computes

their values. Together the three must include inst, addr and treg, since the picture shows those three names on those three outputs to other blocks. The only question is *which* outputs to associate with which names (and whether indeed it matters).

In this case, the issue is helpfully resolved by the types of the three signals: out7 has type word12, so it must correspond to inst_sig, whose type we *know* to be word12 from Figure 1 (which, had space pemitted, would have included the types of inter-block lines); and so on. Thus we get the apparently 'reversed ordering' of out5 and out7 in Figure 2.

The point of all this is that the implicit semantics of pictures carry no ordering information; they just tell us that REGISTERS_COMB has nineteen arguments altogether: twelve signals corresponding to the twelve input values plus the seven output values of the function REGISTERS. The textual representation, by its nature, *must* order arguments: out7 is the *seventh* output of REGISTERS and out7_sig is the *nineteenth* argument of REGISTERS_COMB. However, there is no formal indication in the text that inst is intended to be the seventh output value of REGISTERS[17]. Without the fortuitous aid of the distinctness of the types of out5, out6 and out7, these three outputs could be paired off in any fashion with the signals inst, addr and treg. The pairing *does* in fact matter, because it could result in the seventh argument being computed by the function CLOCK_ADDR or CLOCK_REGT (see appendix for full definitions) rather than by CLOCK_INST, which would be a serious mis-representation of the 'intended' design. Since the intention is not formally reported, this demonstrates how a picture and its text together may not necessarily determine a correct representation of the designers' intentions. Notational tricks may allow us to make intelligent guesses, but there is no way to be *certain* that a derived representation is correct with respect to intention. This is an important point which is also mentioned in Section 2 and Section 10.

As for the other four outputs of the latch, type information only enables the p-register line to be identified (with out4); the other three are pictorially interchangeable as long as we capture the intended computations by pairing out1 with areg, out2 with xreg and out3 with yreg. Again, nothing supplied formally specifies the intended pairings. In drawing the picture we can choose arbitrarily, since the picture conveys no ordering information. **(This ends the Aside.)**

Next, a consequence of the definition of REGISTERS_COMB is inferred which gives a more convenient form of the definition for reasoning purposes. The consequence

[17]The original RSRE text is annotated between lines with comments to this effect, but that, of course, is not a formal indication either.

requires the introduction of new HOL functions called SECOND, THIRD, and so on, for selecting elements of ordered tuples. (FIRST already exists in HOL as FST, and SEVENTH' differs from SEVENTH in choosing the seventh and final rather the seventh of eight or more elements of a tuple.) From the definition of REGISTERS_COMB, it follows from simple properties of pairing that

```
|- REGISTERS_COMB
   (extdatabar_sig,aoutbar_sig,clkenb_sig,regstrobe_sig,strobeaddr_sig,
    areg_sig,xreg_sig,yreg_sig,preg_sig,treg_sig,addr_sig,inst_sig,
    out1_sig,out2_sig,out3_sig,out4_sig,out5_sig,out6_sig,out7_sig) =
   (!n.
     (out1_sig n =
      FST
      (REGISTERS
       (extdatabar_sig n,aoutbar_sig n,clkenb_sig n,regstrobe_sig n,strobeaddr_sig n,
        areg_sig n,xreg_sig n,yreg_sig n,preg_sig n,treg_sig n,addr_sig n,inst_sig n))) /\
     (out2_sig n =
      SECOND
      (REGISTERS
       (extdatabar_sig n,aoutbar_sig n,clkenb_sig n,regstrobe_sig n,strobeaddr_sig n,
        areg_sig n,xreg_sig n,yreg_sig n,preg_sig n,treg_sig n,addr_sig n,inst_sig n))) /\
     (out3_sig n = ...) /\
     (out4_sig n = ...) /\
     (out5_sig n = ...) /\
     (out6_sig n = ...) /\
     (out7_sig n =
      SEVENTH'
      (REGISTERS
       (extdatabar_sig n,aoutbar_sig n,clkenb_sig n,regstrobe_sig n,strobeaddr_sig n,
        areg_sig n,xreg_sig n,yreg_sig n,preg_sig n,treg_sig n,addr_sig n,inst_sig n))))
```

That is, the *relation* holds over the signals shown if and only if at all times the *function* takes inputs and gives outputs as shown. Note that there is still no advance in time; i.e., this is still a fact only about a combinational part of the DATAREG block.

To specify the whole DATAREG block, which *does* involve an advance in time, we might first try to describe a relation (called DATAREG) in terms of nine already-defined relations:

```
|- DATAREG
   (extdatabar_sig,aoutbar_sig,clkenb_sig,regstrobe_sig,strobeaddr_sig,rsel_sig,rbar_sig,
    treg_sig,addr_sig,inst_sig) =
   (?out1_sig out2_sig out3_sig out4_sig out5_sig out6_sig out7_sig
     areg_sig xreg_sig yreg_sig preg_sig.
     REGISTERS_COMB
     (extdatabar_sig,aoutbar_sig,clkenb_sig,regstrobe_sig,strobeaddr_sig,
      areg_sig,xreg_sig,yreg_sig,preg_sig,treg_sig,addr_sig,inst_sig,
      out1_sig,out2_sig,out3_sig,out4_sig,out5_sig,out6_sig,out7_sig) /\
     LATCH32(out1_sig,areg_sig) /\
     LATCH32(out2_sig,xreg_sig) /\
     LATCH32(out3_sig,yreg_sig) /\
     LATCH20(out4_sig,preg_sig) /\
     LATCH32(out5_sig,treg_sig) /\
     LATCH20(out6_sig,addr_sig) /\
     LATCH12(out7_sig,inst_sig) /\
     REGSELECT_COMB(areg_sig,xreg_sig,yreg_sig,preg_sig,rsel_sig,rbar_sig))
```

That is, the new relation DATAREG would hold of the ten input/output signals of the whole block if and only if there could be found eleven internal signals (appropriately typed) such that all nine relations held as shown – the two computation relations as well as the seven latch relations.

This in itself is perfectly correct, but it turns out not to be useful. The ultimate aim is to 'solve for' the internal variables (i.e. to hide them), and hence derive a relation holding over just the visible state (external variables) of the block. To do this (at least with existing HOL tools), each such variable must have an associated equation of suitable form. For example, for `out1_sig n` we know (from the consequence of the definition of REGISTERS_COMB) that

```
out1_sig n =
FST
(REGISTERS
 (extdatabar_sig n,aoutbar_sig n,clkenb_sig n,regstrobe_sig n,strobeaddr_sig n,
  areg_sig n,xreg_sig n,yreg_sig n,preg_sig n,treg_sig n,addr_sig n,inst_sig n))
```

which provides a way to solve for `out1_sig` – by replacing it with another expression not involving it. However, in the case of `areg_sig`, the latch relations tell us only that

```
|- LATCH32(out1_sig,areg_sig) = (!n. areg_sig(n+1) = out1_sig n)
```

which is not adequate. The syntactic reason it is not adequate is that we have no equation for `areg_sig` at time `n` with which to expand; only at time `n+1`. The intuitive reason it is not adequate is that the a-register (as well as the p-, x- and y-registers) is really a memory element; it 'remembers' its previous values, and so is genuinely part of the *overall state* of the block machine; it should (indeed must) be given explicitly in the overall block level state, and in the overall state of DATAREG. We therefore describe the whole DATAREG block as a relation by writing:

```
|- DATAREG
   (extdatabar_sig,aoutbar_sig,clkenb_sig,regstrobe_sig,strobeaddr_sig,rsel_sig,
    areg_sig,xreg_sig,yreg_sig,preg_sig,
    rbar_sig,
    treg_sig,addr_sig,inst_sig) =
   (?out1_sig out2_sig out3_sig out4_sig out5_sig out6_sig out7_sig.
     REGISTERS_COMB
     (extdatabar_sig,aoutbar_sig,clkenb_sig,regstrobe_sig,strobeaddr_sig,
      areg_sig,xreg_sig,yreg_sig,preg_sig,treg_sig,addr_sig,inst_sig,
      out1_sig,out2_sig,out3_sig,out4_sig,out5_sig,out6_sig,out7_sig) /\
     LATCH32(out1_sig,areg_sig) /\
     LATCH32(out2_sig,xreg_sig) /\
     LATCH32(out3_sig,yreg_sig) /\
     LATCH20(out4_sig,preg_sig) /\
     LATCH32(out5_sig,treg_sig) /\
     LATCH20(out6_sig,addr_sig) /\
     LATCH12(out7_sig,inst_sig) /\
     REGSELECT_COMB(areg_sig,xreg_sig,yreg_sig,preg_sig,rsel_sig,rbar_sig))
```

Here, the relation DATAREG has four more explicit signal parameters than in the first attempt; otherwise it is the same. This definition (a mutually recursive set of equations) can be repeatedly unfolded, with replacements made where possible based on (i) the consequence of the definition of REGISTERS_COMB, and (ii) the definitions of the nine relations, to solve for the seven internal variables. This is a standard technique, commonly used in modelling hardware in HOL. A general ML procedure

encodes the inference patterm in HOL. It gives the following result, asserting that the DATAREG relation holds if and only if eight facts hold respectively about the eight state variables (four registers and four outputs) of the block:

```
|- DATAREG
   (extdatabar_sig,aoutbar_sig,clkenb_sig,regstrobe_sig,strobeaddr_sig,rsel_sig,
    areg_sig,xreg_sig,yreg_sig,preg_sig,
    rbar_sig,
    treg_sig,addr_sig,inst_sig) =
   (!n.
     areg_sig(n+1) =
     FST
     (REGISTERS
      (extdatabar_sig n,aoutbar_sig n,clkenb_sig n,regstrobe_sig n,strobeaddr_sig n,
       areg_sig n,xreg_sig n,yreg_sig n,preg_sig n,treg_sig n,addr_sig n,inst_sig n))) /\
   (!n.
     xreg_sig(n+1) =
     SECOND
     (REGISTERS
      (extdatabar_sig n,aoutbar_sig n,clkenb_sig n,regstrobe_sig n,strobeaddr_sig n,
       areg_sig n,xreg_sig n,yreg_sig n,preg_sig n,treg_sig n,addr_sig n,inst_sig n))) /\
   (!n.
     yreg_sig(n+1) = ...) /\
   (!n.
     preg_sig(n+1) = ...) /\
   (!n.
     treg_sig(n+1) = ...) /\
   (!n.
     addr_sig(n+1) = ...) /\
   (!n.
     inst_sig(n+1) =
     SEVENTH'
     (REGISTERS
      (extdatabar_sig n,aoutbar_sig n,clkenb_sig n,regstrobe_sig n,strobeaddr_sig n,
       areg_sig n,xreg_sig n,yreg_sig n,preg_sig n,treg_sig n,addr_sig n,inst_sig n))) /\
   (!n.
     rbar_sig n =
     REGSELECT(areg_sig n,xreg_sig n,yreg_sig n,preg_sig n,rsel_sig n))
```

Note that the constructed relations REGISTERS_COMB and REGSELECT_COMB have disappeared in the end result, and the expression for the DATAREG relation is now in terms of the original functions REGISTERS and REGSELECT. The constructed relations were only devices for deriving the expression for the DATAREG relation. Ultimately, DATAREG itself will turn out to be just a device in deriving the expression representing the whole block model.

We now have expressions for the four outputs of the DATAREG block, as well as for the four internal state values (the a-, x-, y- and p- registers) of the block. These expressions do, finally, reflect the time delay caused by the latch: in each conjunct but the last, the function REGISTERS is applied to a time-n input to yield a time-(n+1) output. In the last conjunct, the block output rbar is not latched, so its expression does not reflect an advance in time. These eight expressions, conjoined, characterize the relation DATAREG.

The other blocks are treated similarly, with relations defined as intermediate devices where required. Some blocks have latches and some do not. The BANDSTOP block, like DATAREG does – two internal lines are hidden. BANDSTOP_BLOCK has six inputs and two outputs. The inputs, in order, are: two control signals (stmplx_sig and bmplx_sig) from

the decoder, to control the stopping flag and the boolean flag, respectively; the 4-bit function selector fsf_sig (projected out from the 12-bit instruction signal); a timing signal regstrobe_sig, from the timing block; another control signal (clkenbp_sig) from the decoder, indicating whether the program counter is the chosen destination (in which case values with more than 20 significant bits must be caught so that the stopping flag is set); and a 9-bit signal (conditions_sig) from the arithmetic-logic unit, coding information about the computed result and the selected destination. The outputs, both latched, are just the boolean flag signal and the stopping flag signal. The main function of the block is the given function BANDSTOP. (SND, below, is the existing HOL function for projecting the second element of a pair.)

```
|- BANDSTOP_BLOCK
   (stmplx_sig,bmplx_sig,fsf_sig,regstrobe_sig,clkenbp_sig,conditions_sig,stop_sig,bflag_sig) =
   (!n. stop_sig(n+1) =
        FST
        (BANDSTOP
         (stmplx_sig n,bmplx_sig n,fsf_sig n,regstrobe_sig n,clkenbp_sig n,conditions_sig n,
          stop_sig n,bflag_sig n))) /\
   (!n. bflag_sig(n+1) =
        SND
        (BANDSTOP
         (stmplx_sig n,bmplx_sig n,fsf_sig n,regstrobe_sig n,clkenbp_sig n,conditions_sig n,
          stop_sig n,bflag_sig n)))
```

The DECODER block takes the (12-bit) instruction signal, the (coded) major state signal, the boolean flag signal and the stopping flag signal. It has ten output signals, all latched except two: an unlatched indicator (cond_sig) of the next major state when there is a choice of next major state; and an unlatched indicator (call_sig) that a procedure call is being processed. The latched signals are: a control signal (alucon_sig) to the arithmetic-logic unit, carrying information on the operation to be performed; bmplx_sig and stmplx_sig as above; an indicator (carryused_sig) of a carry, to the timing block; a boolean-valued signal (cin_sig) of the carry-in (which doubles as the least significant bit in some left shifts); and clkenb_sig, clkenbp_sig and rsel_sig as above. The main function of the block is INST_DECODER.

```
|- DECODER
   (inst_sig,major_sig,bflag_sig,stop_sig,cond_sig,call_sig,alucon_sig,bmplx_sig,
    carryused_sig,cin_sig,clkenb_sig,clkenbp_sig,rsel_sig,stmplx_sig) =
   (!n. bmplx_sig(n+1) = FST(INST_DECODER(inst_sig n,major_sig n,bflag_sig n,stop_sig n))) /\
   (!n. stmplx_sig(n+1) = SECOND(INST_DECODER(inst_sig n,major_sig n,bflag_sig n,stop_sig n))) /\
   (!n. rsel_sig(n+1) = THIRD(INST_DECODER(inst_sig n,major_sig n,bflag_sig n,stop_sig n))) /\
   (!n. cin_sig(n+1) = FOURTH(INST_DECODER(inst_sig n,major_sig n,bflag_sig n,stop_sig n))) /\
   (!n. alucon_sig(n+1) = FIFTH(INST_DECODER(inst_sig n,major_sig n,bflag_sig n,stop_sig n))) /\
   (!n. carryused_sig(n+1) = SIXTH(INST_DECODER(inst_sig n,major_sig n,bflag_sig n,stop_sig n))) /\
   (!n. clkenb_sig(n+1) = SEVENTH(INST_DECODER(inst_sig n,major_sig n,bflag_sig n,stop_sig n))) /\
   (!n. clkenbp_sig(n+1) = EIGHTH(INST_DECODER(inst_sig n,major_sig n,bflag_sig n,stop_sig n))) /\
   (!n. cond_sig n = NINTH(INST_DECODER(inst_sig n,major_sig n,bflag_sig n,stop_sig n))) /\
   (!n. call_sig n = TENTH(INST_DECODER(inst_sig n,major_sig n,bflag_sig n,stop_sig n)))
```

Likewise, the MAJOR and MINOR blocks have a latch each. MAJOR takes the stopping signal; call_sig as above; an indication (timeoutbar_sig) of a time-out; two timing signals (nextmnbar_sig and advance_sig); a signal indicating a reset (reset_sig); and cond_sig

as above. The latched (main) output is the major state signal, major_sig. The unlatched output (intresetbar_sig) is an indication, sent to MINOR, of an error (so that MINOR can stop counting the sub-states of the major state and re-initialize). The main function of MAJOR is MAJORLOGIC.

```
|- MAJOR
   (stop_sig,call_sig,timeoutbar_sig,nextmnbar_sig,advance_sig,reset_sig,cond_sig,
    major_sig,intresetbar_sig) =
   (!n. major_sig(n+1) =
        FST
        (MAJORLOGIC
         (stop_sig n,call_sig n,timeoutbar_sig n,nextmnbar_sig n,advance_sig n,reset_sig n,
          cond_sig n,major_sig n))) /\
   (!n. intresetbar_sig n =
        SND
        (MAJORLOGIC
         (stop_sig n,call_sig n,timeoutbar_sig n,nextmnbar_sig n,advance_sig n,reset_sig n,
          cond_sig n,major_sig n)))
```

MINOR takes the same two timing signals (nextmnbar_sig and advance_sig), and reset_sig and intresetbar_sig as above. It has only one output (latched) – the minor state signal. The main function of the block is NEXTMINOR.

```
|- MINOR(nextmnbar_sig,advance_sig,reset_sig,intresetbar_sig,minor_sig) =
   (!n. minor_sig(n+1) =
        NEXTMINOR(nextmnbar_sig n,advance_sig n,reset_sig n,intresetbar_sig n,minor_sig n))
```

The external interface, EXTERNAL_BLOCK, takes five signals from the internal part of the machine, one from the memory, and the four signals from the outside world; it returns one signal to the internal machine, ten to either the memory or the outside world, and four latched signals to the internal machine. The only sub-function is EXTERNAL, which computes the data signal, extdatabar, to the circuit; the four signals to the latch; and the ten signals *not* to the internal circuit.

```
|- EXTERNAL_BLOCK
   (rbar_sig,addr_sig,bflag_sig,strobe_sig,major_sig,e_data_in_sig,
    e_resetbar_sig,e_errorbar_sig,e_stepbar_sig,e_reply_sig,extdatabar_sig,
    e_bflag_sig,e_fetchbar_sig,e_iobar_sig,e_perform_sig,e_data_out_sig,e_address_sig,
    e_stopped_sig,e_majorstate_sig,e_strobebar_sig,e_writebar_sig,
    error_sig,pause_sig,reply_sig,reset_sig) =
   (!n. extdatabar_sig n =
        FST
        (EXTERNAL
         (rbar_sig n,addr_sig n,bflag_sig n,strobe_sig n,major_sig n,e_data_in_sig n,
          e_resetbar_sig n,e_errorbar_sig n,e_stepbar_sig n,e_reply_sig n))) /\
   (!n. reset_sig(n+1) =
        SECOND
        (EXTERNAL
         (rbar_sig n,addr_sig n,bflag_sig n,strobe_sig n,major_sig n,e_data_in_sig n,
          e_resetbar_sig n,e_errorbar_sig n,e_stepbar_sig n,e_reply_sig n))) /\
   (!n. error_sig(n+1) = ...) /\
   (!n. pause_sig(n+1) = ...) /\
   (!n. reply_sig(n+1) = ...) /\
   (!n. e_data_out_sig n = ...) /\
   (!n. e_address_sig n = ...) /\
   (!n. e_bflag_sig n = ...) /\
   (!n. e_majorstate_sig n = ...) /\
   (!n. e_strobebar_sig n = ...) /\
   (!n. e_stopped_sig n = ...) /\
   (!n. e_perform_sig n = ...) /\
   (!n. e_fetchbar_sig n = ...) /\
   (!n. e_iobar_sig n = ...) /\
   (!n. e_writebar_sig n =
     FIFTEENTH
     (EXTERNAL
      (rbar_sig n,addr_sig n,bflag_sig n,strobe_sig n,major_sig n,e_data_in_sig n,
       e_resetbar_sig n,e_errorbar_sig n,e_stepbar_sig n,e_reply_sig n)))
```

The memory, MEMORY_BLOCK, takes the five signals from the external interface and returns one signal to it: (e_data_in_sig). There is one latched signal, ram_sig, purely internal to the memory, representing the state of the memory. The only subfunction is MEMORY, which computes a new memory and data output based on the old memory and the inputs.

```
|- MEMORY_BLOCK
   (e_data_out_sig,e_address_sig,e_iobar_sig,e_writebar_sig,e_strobebar_sig,ram_sig,
    e_data_in_sig) =
   (!n.
     ram_sig(n+1) =
     FST
     (MEMORY
      (ram_sig n,
       e_data_out_sig n,e_address_sig n,e_iobar_sig n,e_writebar_sig n,e_strobebar_sig n))) /\
   (!n.
     e_data_in_sig n =
     SND
     (MEMORY
      (ram_sig n,
       e_data_out_sig n,e_address_sig n,e_iobar_sig n,e_writebar_sig n,e_strobebar_sig n)))
```

Blocks without latches are simpler. For example, as Figure 1 indicates, ALU_COMB has five inputs (two of them *from* DATAREG) and two outputs (one of them *to* DATAREG). The inputs are: rbar_sig, the data signal to the external interface; treg_sig from DATAREG; cin_sig as above; the boolean flag; and alucon_sig as above. The outputs are: the computed data output aoutbar_sig to DATAREG (as above); and a 9-bit word, conditions_sig, encoding certain facts, to BANDSTOP, as above. Without latches there are no internal lines to hide. The main function of the block is ALU.

```
|- ALU_COMB
   (rbar_sig,treg_sig,cin_sig,bflag_sig,alucon_sig,aoutbar_sig,conditions_sig) =
   (!n.
     (aoutbar_sig n = FST(ALU(rbar_sig n,treg_sig n,cin_sig n,bflag_sig n,alucon_sig n))) /\
     (conditions_sig n = SND(ALU(rbar_sig n,treg_sig n,cin_sig n,bflag_sig n,alucon_sig n))))
```

Similarly, the TIMING block has no latches. In order, it takes major_sig and minor_sig as above; two signals, pause_sig and reply_sig, from the external interface; and carryused_sig as above. It outputs five timing signals to various blocks: advance_sig, nextmnbar_sig, regstrobe_sig, strobe_sig and strobeaddr_sig. (See [22] for explanations of these.) The main function of the block is TIMING.

```
|- TIMING_COMB
   (major_sig,minor_sig,pause_sig,reply_sig,carryused_sig,advance_sig,nextmnbar_sig,
    regstrobe_sig,strobe_sig,strobeaddr_sig) =
   (!n.
     (advance_sig n =
      FST(TIMING(major_sig n,minor_sig n,pause_sig n,reply_sig n,carryused_sig n))) /\
     (nextmnbar_sig n =
      SECOND(TIMING(major_sig n,minor_sig n,pause_sig n,reply_sig n,carryused_sig n))) /\
     (regstrobe_sig n =
      THIRD(TIMING(major_sig n,minor_sig n,pause_sig n,reply_sig n,carryused_sig n))) /\
     (strobe_sig n =
      FOURTH(TIMING(major_sig n,minor_sig n,pause_sig n,reply_sig n,carryused_sig n))) /\
     (strobeaddr_sig n =
      FIFTH'(TIMING(major_sig n,minor_sig n,pause_sig n,reply_sig n,carryused_sig n))))
```

The other blocks are not necessary for this exposition; we just point out that the model includes exactly two other purely internal registers which like areg and so on must be included as state variables. The first is ram, the 32-bit memory value in MEMORY_BLOCK, and the second is count in TIMEOUT_BLOCK, a 6-bit counter which at its maximum value indicates that the machine should be timed out (due to a memory failure).

To summarize this section, the DATAREG block was used as an example of the method of deriving a formal expression (a relation) to describe a whole block which is composed of sub-blocks and latches. We began with what was given: the functional definitions of the two sub-blocks REGISTERS and REGSELECT, as well as the picture shown in Figure 2, without names on the internal lines. The notion of signals – functions which produce values given times – was introduced. We discussed why it was necessary to identify internal lines with arguments of REGISTERS, and how these pairings were not fully determined by the combination of picture and text, thus introducing the possibility of incorrectly representing the design of the machine. We then extracted the notion of the *state* of the whole DATAREG block, and used that to infer a description of the whole block in terms of state signals only, with internal lines concealed. All of the blocks in the model are treated similarly. In subsequent sections, the very same method is used to describe the entire block diagram by a relation, taking the ten blocks as the basic units.

6.3.2 Joining Blocks

The ultimate aim in this section is to derive a logical expression representing the whole block model, both its pictorial and the textual parts. The ten block are first represented as relations, just as we have represented DATAREG. They are then combined using exactly the same method as we used to combine the sub-units of the DATAREG block itself. Internal lines which are not state (latch output) values are hidden when possible, as the out_i were hidden. The units of the block model can be combined in any groupings or all at once. Just as an example, suppose we combine just two: DATAREG and ALU_COMB. (These are both discussed in the previous section.)

To represent the combination, we define a relation, called, say DATAREG_ALU, which applies to the lines to and from DATAREG, and to the lines to and from ALU_COMB. (See Figure 1 for a picture of this combination.) Expanding the resulting expression, it is possible to solve for two of the lines internal to the *combined* block: rbar and aoutbar. However, treg is latched in the combined block. This gives expressions for the seven registers (including treg) and for the ALU_COMB output conditions. The

resulting a-register expression, for example, is now

```
(!n.
  areg_sig(n+1) =
  FST
  (REGISTERS
   (extdatabar_sig n,
    FST
    (ALU
     (REGSELECT(areg_sig n,xreg_sig n,yreg_sig n,preg_sig n,rsel_sig n),
      treg_sig n,cin_sig n,bflag_sig n,alucon_sig n)),
    clkenb_sig n,regstrobe_sig n,strobeaddr_sig n,
    areg_sig n,xreg_sig n,yreg_sig n,preg_sig n,treg_sig n,addr_sig n,inst_sig n)))
```

while conditions has the expression

```
(!n.
  conditions_sig n =
  SND
  (ALU
   (REGSELECT(areg_sig n,xreg_sig n,yreg_sig n,preg_sig n,rsel_sig n),
    treg_sig n,cin_sig n,bflag_sig n,alucon_sig n)))
```

The expression for the second argument of REGISTERS, in the a-register expression, can be compared with the one which appears in the theorem for DATAREG alone (page 41): by now 'aoutbar' in the earlier expression has expanded into an expression involving the function ALU. In addition, an expression (also in terms of ALU) for conditions has emerged.

All ten blocks of the block model are combined, exactly as DATAREG and ALU_COMB have just been combined in the example. When this is achieved, the expanded expression for the whole relation (called EXTERNAL_AND_BLOCK_AND_MEM, for the block with the external interface and the memory included) is seven pages long; it is the beginning of a series of many lengthy expressions in the analysis of the block model[18]. EXTERNAL_AND_BLOCK_AND_MEM holds over four inputs from the outside world (e_resetbar, e_errorbar, e_stepbar and e_reply) as well as thirty-one state variables. (Those names prefixed by 'e_' are lines to/from memory or to/from the outside world. Of the thirty-one state variables, six, namely e_data_in, e_data_out, e_address, e_strobebar, e_writebar and e_iobar, represent lines to/from memory, which are unlatched, and twenty-five are latched variables internal to the block.) It gives expressions for the thirty-one state variables as well as for the five outputs (e_bflag, e_majorstate, e_stopped, e_perform and e_fetchbar) of the block model. (The forty lines can all be found in Figure 1; some were introduced in Section 6.3. See [22] for further explanations of the lines.) All other lines shown in Figure 1 are hidden.

[18]The seven pages are formatted and pretty-printed in standard HOL output style

```
|- EXTERNAL_AND_BLOCK_AND_MEM
   (e_data_out_sig,e_address_sig,e_strobebar_sig,e_writebar_sig,e_iobar_sig,e_data_in_sig,
    ram_sig,bflag_sig,
    addr_sig,major_sig,pause_sig,reply_sig,reset_sig,error_sig,minor_sig,count_sig,
    alucon_sig,bmplx_sig,carryused_sig,cin_sig,clkenb_sig,clkenbp_sig,rsel_sig,stmplx_sig,
    stop_sig,areg_sig,xreg_sig,yreg_sig,preg_sig,treg_sig,inst_sig,
    e_resetbar_sig,e_errorbar_sig,e_stepbar_sig,e_reply_sig,e_bflag_sig,e_fetchbar_sig,
    e_perform_sig,e_stopped_sig,e_majorstate_sig) =
   (!n.
     areg_sig(n+1) =
     FST
     (REGISTERS
      (FST
       (EXTERNAL
        (REGSELECT
         (areg_sig n,xreg_sig n,yreg_sig n,preg_sig n,rsel_sig n),
         addr_sig n,bflag_sig n,
         FOURTH
         (TIMING
          (major_sig n,minor_sig n,pause_sig n,reply_sig n,carryused_sig n),
        major_sig n,e_data_in_sig n,e_resetbar_sig n,e_errorbar_sig n,
        e_stepbar_sig n,e_reply_sig n)),
       FST
       (ALU
        (REGSELECT
         (areg_sig n,xreg_sig n,yreg_sig n,preg_sig n,rsel_sig n),
         treg_sig n,cin_sig n,bflag_sig n,alucon_sig n)),
       clkenb_sig n,
       THIRD
       (TIMING
        (major_sig n,minor_sig n,pause_sig n,reply_sig n,carryused_sig n)),
       FIFTH'
       (TIMING
        (major_sig n,minor_sig n,pause_sig n,reply_sig n,carryused_sig n)),
       areg_sig n,xreg_sig n,yreg_sig n,preg_sig n,treg_sig n,addr_sig n,inst_sig n))) /\
   .
   .
   .
```

Comparing this expression for the a-register with that on page 41 gives some idea of the evolution of the relation EXTERNAL_AND_BLOCK_AND_MEM, and some idea of how the inter-connectedness of the block model is expressed. There is a similar expression, of course, for each of the state variables and outputs. For brevity, we abbreviate each of these expressions by defining a constant to stand for it. For example, we define

```
AREG_ABBR
(treg,areg,xreg,yreg,preg,rsel,addr,bflag,major,minor,pause,reply,carryused,
 e_data_in,e_resetbar,e_errorbar,e_stepbar,e_reply,cin,alucon,clkenb,inst)
```

to stand for the final a-register expression. AREG_ABBR terms can then be unfolded or folded as required. It is helpful with very long expressions to shorten them for as long as possible in the development.

In a similar way, expressions are derived and abbreviations defined for all thirty-one registers and five outputs. Some of the key register expressions are shown below: minor, major, bflag, stop, alucon and carryused. The 1-bit carryused value, sent from the decoder to the timing block indicates a carry, and the 7-bit alucon value, from the decoder to the ALU carries information on the ALU operation to be performed. As mentioned in Section 6.3.1, the main sub-functions of the MAJOR, MINOR and DECODER blocks, respectively, are MAJORLOGIC, NEXTMINOR and INST_DECODER.

```
|- MINOR_ABBR
   (minor,major,pause,reply,carryused,reset,stop,inst,bflag,count,error) =
   NEXTMINOR
   (SECOND(TIMING(major,minor,pause,reply,carryused)),
    FST(TIMING(major,minor,pause,reply,carryused)),reset,
    SND
    (MAJORLOGIC
     (stop,TENTH(INST_DECODER(inst,major,bflag,stop)),
      SND
      (TIMEOUT
       (count,reset,error,
        FOURTH(TIMING(major,minor,pause,reply,carryused)))),
      SECOND(TIMING(major,minor,pause,reply,carryused)),
      FST(TIMING(major,minor,pause,reply,carryused)),reset,
      NINTH(INST_DECODER(inst,major,bflag,stop)),major)),minor)

|- MAJOR_ABBR
   (major,stop,inst,bflag,count,reset,error,minor,pause,reply,carryused) =
   FST
   (MAJORLOGIC
    (stop,TENTH(INST_DECODER(inst,major,bflag,stop)),
     SND
     (TIMEOUT
      (count,reset,error,
       FOURTH(TIMING(major,minor,pause,reply,carryused)))),
     SECOND(TIMING(major,minor,pause,reply,carryused)),
     FST(TIMING(major,minor,pause,reply,carryused)),reset,
     NINTH(INST_DECODER(inst,major,bflag,stop)),major))

|- BFLAG_ABBR
   (stop,stmplx,bmplx,inst,major,minor,pause,reply,carryused,clkenbp,areg,xreg,yreg,
    preg,rsel,treg,cin,bflag,alucon) =
   SND
   (BANDSTOP
    (stmplx,bmplx,FSELECT inst,
     THIRD(TIMING(major,minor,pause,reply,carryused)),clkenbp,
     SND(ALU(REGSELECT(areg,xreg,yreg,preg,rsel),treg,cin,bflag,alucon)),stop,bflag))

|- STOP_ABBR
   (bflag,stmplx,bmplx,inst,major,minor,pause,reply,carryused,clkenbp,areg,xreg,yreg,
    preg,rsel,treg,cin,alucon,stop) =
   FST
   (BANDSTOP
    (stmplx,bmplx,FSELECT inst,
     THIRD(TIMING(major,minor,pause,reply,carryused)),clkenbp,
     SND(ALU(REGSELECT(areg,xreg,yreg,preg,rsel),treg,cin,bflag,alucon)),stop,bflag))

|- ALUCON_ABBR(inst,major,bflag,stop) = FIFTH(INST_DECODER(inst,major,bflag,stop))

|- CARRYUSED_ABBR(inst,major,bflag,stop) = SIXTH(INST_DECODER(inst,major,bflag,stop))
```

These and the other thirty-five expressions help us to derive what we ultimately want: a *function* respresenting the block model. The function (called WHOLE_BLOCK_NEXT because it computes the next state of the block) applies to the thirty-one state variables and the four inputs. It returns new values for the thirty-one state variables and also for the five outputs to the outside world. The form of the function is:

```
|- WHOLE_BLOCK_NEXT
   (areg,xreg,yreg,preg,treg,addr,inst,error,pause,reply,reset,
    e_iobar,e_data_out,e_address,e_strobebar,e_writebar,ram,e_data_in,
    minor,major,bflag,stop,count,alucon,bmplx,carryused,cin,clkenb,clkenbp,rsel,stmplx,
    e_resetbar,e_errorbar,e_stepbar,e_reply) =
     let e_bflag = ... in let e_majorstate = ... in let e_stopped = ... in
     let e_perform = ... in let e_fetchbar = ... in
     let areg' = AREG_ABBR
                 (treg,areg,xreg,yreg,preg,rsel,addr,bflag,major,minor,
                  pause,reply,carryused,
                  e_data_in,e_resetbar,e_errorbar,e_stepbar,e_reply,
                  cin,alucon,clkenb,inst) in
     let xreg' = ... in let yreg' = ... in let preg' = ... in let treg' = ... in
     let addr' = ... in let inst' = ... in
     let error' = ... in let pause' = ... in let reply' = ... in let reset' = ... in
     let e_iobar' = ... in let e_data_out' = ... in let e_address' = ... in
     let e_strobebar' = ... in let e_writebar' = ... in
     let ram' = ... in let e_data_in' = ... in
     let minor' = MINOR_ABBR
                  (minor,major,pause,reply,carryused,reset,stop,inst,bflag,count,error) in
     let major' = MAJOR_ABBR
                  (major,stop,inst,bflag,count,reset,error,minor,pause,reply,carryused) in
     let bflag' = BFLAG_ABBR
                  (stop,stmplx,bmplx,inst,major,minor,pause,reply,carryused,clkenbp,
                   areg,xreg,yreg,preg,rsel,treg,cin,bflag,alucon) in
     let stop' = STOP_ABBR
                 (bflag,stmplx,bmplx,inst,major,minor,pause,reply,carryused,clkenbp,
                  areg,xreg,yreg,preg,rsel,treg,cin,alucon,stop) in
     let count' = ... in
     let alucon' = ALUCON_ABBR(inst,major,bflag,stop) in
     let bmplx' = ... in
     let carryused' = CARRYUSED_ABBR(inst,major,bflag,stop) in
     let cin' = ... in let clkenb' = ... in let clkenbp' = ... in
     let rsel' = ... in let stmplx' = ... in
     areg',xreg',yreg',preg',treg',addr',inst',error',pause',reply',reset',
     e_iobar',e_data_out',e_address',e_strobebar',e_writebar',
     ram',e_data_in',minor',major',bflag',stop',count',alucon',bmplx',carryused',cin',
     clkenb',clkenbp',rsel',stmplx',
     e_bflag,e_majorstate,e_stopped,e_perform,e_fetchbar
```

Since the function was constructed from the text of another expression (that of the relation EXTERNAL_AND_BLOCK_AND_MEM), it remains to confirm that the function correctly represents the relation. That is, we prove:

```
|- (!n.
     areg_sig(n+1),xreg_sig(n+1),yreg_sig(n+1),preg_sig(n+1),treg_sig(n+1),
     addr_sig(n+1),inst_sig(n+1),
     error_sig(n+1),pause_sig(n+1),reply_sig(n+1),reset_sig(n+1),
     e_iobar_sig n,e_data_out_sig n,e_address_sig n,e_strobebar_sig n,
     e_writebar_sig n,
     ram_sig(n+1),e_data_in_sig n,
     minor_sig(n+1),major_sig(n+1),bflag_sig(n+1),stop_sig(n+1),count_sig(n+1),
     alucon_sig(n+1),bmplx_sig(n+1),carryused_sig(n+1),cin_sig(n+1),
     clkenb_sig(n+1),clkenbp_sig(n+1),rsel_sig(n+1),stmplx_sig(n+1),
     e_bflag_sig n,e_majorstate_sig n,e_stopped_sig n,e_perform_sig n,e_fetchbar_sig n =
     WHOLE_BLOCK_NEXT
     (areg_sig n,xreg_sig n,yreg_sig n,preg_sig n,treg_sig n,addr_sig n,inst_sig n,
      error_sig n,pause_sig n,reply_sig n,reset_sig n,
      e_iobar_sig n,e_data_out_sig n,e_address_sig n,e_strobebar_sig n,e_writebar_sig n,
      ram_sig n,e_data_in_sig n,
      minor_sig n,major_sig n,bflag_sig n,stop_sig n,count_sig n,alucon_sig n,
      bmplx_sig n,carryused_sig n,cin_sig n,clkenb_sig n,clkenbp_sig n,rsel_sig n,stmplx_sig n,
      e_resetbar_sig n,e_errorbar_sig n,e_stepbar_sig n,e_reply_sig n)) =

   EXTERNAL_AND_BLOCK_AND_MEM
   (e_data_out_sig,e_address_sig,e_strobebar_sig,e_writebar_sig,e_iobar_sig,e_data_in_sig,
    ram_sig,bflag_sig,addr_sig,major_sig,
    pause_sig,reply_sig,reset_sig,error_sig,
    minor_sig,count_sig,alucon_sig,bmplx_sig,carryused_sig,cin_sig,clkenb_sig,clkenbp_sig,
    rsel_sig,stmplx_sig,
    stop_sig,areg_sig,xreg_sig,yreg_sig,preg_sig,treg_sig,inst_sig,
    e_resetbar_sig,e_errorbar_sig,e_stepbar_sig,e_reply_sig,e_bflag_sig,e_fetchbar_sig,
    e_perform_sig,e_stopped_sig,e_majorstate_sig)
```

This means that the function WHOLE_BLOCK_NEXT maps time-n values of states and inputs

to the values of states and outputs as shown[19] (that is, it sequences values correctly in time) **if and only if** the relation EXTERNAL_AND_BLOCK_AND_MEM holds over input, state and output *signals*. The theorem follows more-or-less immediately from the definitions of the function and the relation, and from properties of pairing. That in turn implies that each of the state and output values can be computed by the (abbreviated) expression derived for it:

Sequencing Theorem

```
|- (!n.
     areg_sig(n+1),xreg_sig(n+1),yreg_sig(n+1),preg_sig(n+1),treg_sig(n+1),
     addr_sig(n+1),inst_sig(n+1),
     error_sig(n+1),pause_sig(n+1),reply_sig(n+1),reset_sig(n+1),
     e_iobar_sig n,e_data_out_sig n,e_address_sig n,e_strobebar_sig n,e_writebar_sig n,
     ram_sig(n+1),e_data_in_sig n,
     minor_sig(n+1),major_sig(n+1),bflag_sig(n+1),stop_sig(n+1),count_sig(n+1),
     alucon_sig(n+1),bmplx_sig(n+1),carryused_sig(n+1),cin_sig(n+1),
     clkenb_sig(n+1),clkenbp_sig(n+1),rsel_sig(n+1),stmplx_sig(n+1),
     e_bflag_sig n,e_majorstate_sig n,e_stopped_sig n,e_perform_sig n,e_fetchbar_sig n =
     WHOLE_BLOCK_NEXT
     (areg_sig n,xreg_sig n,yreg_sig n,preg_sig n,treg_sig n,addr_sig n,inst_sig n,
      error_sig n,pause_sig n,reply_sig n,reset_sig n,
      e_iobar_sig n,e_data_out_sig n,e_address_sig n,e_strobebar_sig n,e_writebar_sig n,
      ram_sig n,e_data_in_sig n,
      minor_sig n,major_sig n,bflag_sig n,stop_sig n,count_sig n,alucon_sig n,bmplx_sig n,
      carryused_sig n,cin_sig n,clkenb_sig n,clkenbp_sig n,rsel_sig n,stmplx_sig n,
      e_resetbar_sig n,e_errorbar_sig n,e_stepbar_sig n,e_reply_sig n)) ==>
    (areg_sig(n+1) =
     AREG_ABBR
     (treg_sig n,areg_sig n,xreg_sig n,yreg_sig n,preg_sig n,rsel_sig n,addr_sig n,bflag_sig n,
      major_sig n,minor_sig n,pause_sig n,reply_sig n,carryused_sig n,
      e_data_in_sig n,e_resetbar_sig n,e_errorbar_sig n,e_stepbar_sig n,e_reply_sig n,
      cin_sig n,alucon_sig n,clkenb_sig n,inst_sig n) /\
    (xreg_sig(n+1) = ... /\ (yreg_sig(n+1) = ... /\ (preg_sig(n+1) = ... /\
    (treg_sig(n+1) = ... /\ (addr_sig(n+1) = ... /\ (inst_sig(n+1) = ... /\
    (error_sig(n+1) = ... /\ (pause_sig(n+1) = ... /\
    (reply_sig(n+1) = ... /\ (reset_sig(n+1) = ... /\
    (e_iobar_sig n = ... /\ (e_data_out_sig n = ... /\ (e_address_sig n = ... /\
    (e_strobebar_sig n = ... /\ (e_writebar_sig n = ... /\ (ram_sig(n+1) = ... /\
    (e_data_in_sig n = ... /\
    (minor_sig(n+1) =
     MINOR_ABBR
     (minor,major,pause,reply,carryused,reset,stop,inst,bflag,count,error) /\
    (major_sig(n+1) =
     MAJOR_ABBR
     (major,stop,inst,bflag,count,reset,error,minor,pause,reply,carryused) /\
    (bflag_sig(n+1) =
     BFLAG_ABBR
     (stop,stmplx,bmplx,inst,major,minor,pause,reply,carryused,clkenbp,
      areg,xreg,yreg,preg,rsel,treg,cin,bflag,alucon) /\
    (stop_sig(n+1) =
     STOP_ABBR
     (bflag,stmplx,bmplx,inst,major,minor,pause,reply,carryused,clkenbp,
      areg,xreg,yreg,preg,rsel,treg,cin,alucon,stop) /\
    (count_sig(n+1) = ... /\
    (alucon_sig(n+1) = ALUCON_ABBR(inst,major,bflag,stop) in
    (bmplx_sig(n+1) = ... /\
    (carryused_sig(n+1) = CARRYUSED_ABBR(inst,major,bflag,stop) /\
    (cin_sig(n+1) = ... /\ (clkenb_sig(n+1) = ... /\ (clkenbp_sig(n+1) = ... /\
    (rsel_sig(n+1) = ... /\ (stmplx_sig(n+1) = ... /\
    (e_bflag_sig n = ... /\ (e_majorstate_sig n = ... /\ (e_stopped_sig n = ... /\
    (e_perform_sig n = ... /\ (e_fetchbar_sig n = )...)
```

The function WHOLE_BLOCK_NEXT (and in particular the property above) is the essential and basic tool for analyzing the block model of Viper. Its usefulness lies in the

[19]The twenty-five internal state variables are latched so appear at the 'next' time; the five outputs and six lines to/from memory are not latched so appear at the initial time.

fact that since it *is* a function, we can give it inputs and it will compute outputs. These outputs, for latched values, represent the values on lines one time unit later. By repeatedly applying the function, it is possible to describe the behaviour of the block machine during and after the processing of Viper's various instruction types. The rest of this paper describes the analysis of the block model based on the functional expression.

7 Using the Representation: The Minor State Transitions

The analysis of the Viper block machine forms a complex layered structure. Simple facts about the functions (such as DATAREG) representing individual blocks, and about their sub-functions and sub-sub-functions (such as REGISTERS and CLOCK_REGA) support more complex facts about the values on lines (such as the value of areg) at given times. In particular, they support facts about the values of the major and minor lines. This pair of values characterizes the progress of the block machine in its execution of an instruction schema. They are particularly important registers in the analysis because as the function WHOLE_BLOCK_NEXT (representing the block model) is computed successively through major and minor values, results accumulate in various registers, thereby giving a complete description of the behaviour of the block machine – which is the goal. As suggested in the introduction, this stage of the analysis can be viewed at one level as a symbolic execution of the block machine, but supported in addition by the security of formal proof; each state transition is logically *inferred*, and not just computed.

The structure of the analysis is reflected in a hierarchy of HOL theories. So far, for example, we have indicated some of what is contained in the HOL theory about the DATAREG block. Each layer of the subsequent analysis is similarly reflected in an HOL theory. The analysis itself rests on a hierarchy of theories containing lemmas.

One example will suffice for all the cases: we assume that the major state is fixed and equal to 4. That indicates the performance of an ALU operation, specified in detail by the current instruction[20]. As for which ALU operation is chosen in the example, it is not necessary at first to be specific.

[20]In a sequence of major state transitions, an ALU operation would only happen after the major state had been 1, i.e. sometime after a FETCH operation producing the current instruction. In fact, each sequence of major states begins with 1.

7.1 Lemmas about the Thirty-Six Lines

The first step in the analysis is to characterize the minor state transitions of the block machine. That means evaluating the **Sequencing Theorem** (page 50) up to seven times, assuming in turn that the 3-bit minor state counter holds the values 0, 1, 2, and so on[21], while the 4-bit major state remains fixed at 4. Ultimately, these evaluations will permit the composition of the minor state transformations, and so yield the (provably correct) cumulative effect of the *single* transition through the major state 4 (the performance of an ALU operation). The single transition is effected by a recursive procedure which 'runs through' the minor transitions in sequence, accumulating effects on state and output values. (A similar sort of symbolic execution is considered by J. Joyce in [17]).

To infer the effects on the block state of the various minor state transitions, we infer the value of WHOLE_BLOCK_NEXT at these states. This requires deducing the values on thirty-six lines, each described by an abbreviation such as AREG_ABBR. We therefore prove thirty-six sets of lemmas, dubbed '**Class 2**' lemmas because they depend on one further layer of more basic lemmas. For the a-register, for example:

```
VAL3 minor = 0, VAL4 major = 4
|- AREG_ABBR
   (areg,xreg,yreg,preg,rsel,addr,bflag,major,minor,pause,reply,carryused,
    e_data_in,e_resetbar,e_errorbar,e_stepbar,e_reply,treg,cin,alucon,clkenb,inst) =
   areg

VAL3 minor = 1, VAL4 major = 4
|- AREG_ABBR
   (areg,xreg,yreg,preg,rsel,addr,bflag,major,minor,pause,reply,carryused,
    e_data_in,e_resetbar,e_errorbar,e_stepbar,e_reply,treg,cin,alucon,clkenb,inst) =
   areg

VAL3 minor = 2, VAL4 major = 4
|- AREG_ABBR
   (areg,xreg,yreg,preg,rsel,addr,bflag,major,minor,pause,reply,carryused,e_data_in,
    e_resetbar,e_errorbar,e_stepbar,e_reply,treg,cin,alucon,clkenb,inst) =
   areg

VAL3 minor = 3, VAL4 major = 4
|- AREG_ABBR
   (areg,xreg,yreg,preg,rsel,addr,bflag,major,minor,pause,reply,carryused,
    e_data_in,e_resetbar,e_errorbar,e_stepbar,e_reply,treg,cin,alucon,clkenb,inst) =
   ((~EL 0(BITS7 clkenb) /\ ~carryused) =>
    NOT32
    (FST
     (ALU
      (REGSELECT(areg,xreg,yreg,preg,rsel),treg,cin,bflag,alucon))) |
    areg)

VAL3 minor = 4, VAL4 major = 4
|- AREG_ABBR
   (areg,xreg,yreg,preg,rsel,addr,bflag,major,minor,pause,reply,carryused,
    e_data_in,e_resetbar,e_errorbar,e_stepbar,e_reply,treg,cin,alucon,clkenb,inst) =
   ((~EL 0(BITS7 clkenb) /\ carryused) =>
    NOT32
    (FST
     (ALU
      (REGSELECT(areg,xreg,yreg,preg,rsel),treg,cin,bflag,alucon))) |
    areg)
```

[21]Likewise, each sequence of minor states begins with 0. This and the previous footnote are discussed later.

```
VAL3 minor = 5, VAL4 major = 4
|- AREG_ABBR
   (areg,xreg,yreg,preg,rsel,addr,bflag,major,minor,pause,reply,carryused,
    e_data_in,e_resetbar,e_errorbar,e_stepbar,e_reply,treg,cin,alucon,clkenb,inst) =
   areg

VAL3 minor = 6, VAL4 major = 4
|- AREG_ABBR
   (areg,xreg,yreg,preg,rsel,addr,bflag,major,minor,pause,reply,carryused,
    e_data_in,e_resetbar,e_errorbar,e_stepbar,e_reply,
    treg,cin,alucon,clkenb,inst) =
   areg

VAL3 minor = 7, VAL4 major = 4
|- AREG_ABBR
   (areg,xreg,yreg,preg,rsel,addr,bflag,major,minor,pause,reply,carryused,
    e_data_in,e_resetbar,e_errorbar,e_stepbar,e_reply,treg,cin,alucon,clkenb,inst) =
   areg
```

The fortunes of the a-register at each of the eight minor states are now explicit: changes can only occur in the fourth and fifth cycles (minor states 3 and 4), during which, depending on two lines from the decoder, the accumulator *may* take on the (negated) data result (i.e. the first output) of the ALU operation. The relevant information is (i) the least significant bit of the control line `clkenb`, (the bit which controls the loading of the a-register in particular), and (ii) the carry indicator `carryused`.

The eight theorems above are proved by an ML procedure which takes the definition of `AREG_ABBR`, unfolds it on the given values, applies relevant **Class 1** lemmas, and simplifies. This is an example for *forward proof* in HOL (see Section 4); a definition is unfolded and simplified, using lemmas and logical identities, to yield a result which was not necessarily foreseen. All of the many lemmas mentioned or suggested in this section are proved in a similar forward manner.

The form in which the lemmas are shown turns out to be a convenient one: that is, equations with assumptions carried along in the background, rather than implications with equational consequents. With assumptions in the background, substitutions (such as '`4`' for '`VAL4 major`') are made directly, and the assumptions are propagated 'behind the scenes'. In the implicative form (`|- ... /\ (VAL4 major = 4) => ...`), the lemmas would not be useable until the assumptions were dismissed and the substitution(s) made.

Similarly, we prove sets of **Class2** theorems giving the values of the other thirty lines and the five outputs, each at the ten major and eight minor states. As mentioned, the cumulative results for the registers `major` and `minor` are particularly important in the operation of the block machine. These results depend on two lines from the external interface: (i) `reset`, to `MINOR`, `MAJOR` and `TIMEOUT_BLOCK`, which indicates that the machine is to be reset (have its registers cleared, etc), and (ii) `error`, to `TIMEOUT_BLOCK` only, which indicates that the machine is to be stopped because of a

forced error. The results also depend on the 6-bit internal register, count, which indicates a time-out (due to memory failure) when it reaches the value 64. For example, while the major state is fixed at 4 (PERFORM ALU), we prove:

```
VAL3 minor = 0, VAL4 major = 4
|- MAJOR_ABBR
   (major,stop,inst,bflag,count,reset,error,minor,pause,reply,carryused) =
   (reset => #0010 | (((count = #111111) \/ error) => #1000 | #0100))

VAL3 minor = 0, VAL4 major = 4
|- MINOR_ABBR
   (minor,major,pause,reply,carryused,reset,stop,inst,bflag,count,error) =
   ((reset \/ (count = #111111) \/ error) => #000 | #001)
```

That is, for the minor state 0 and major state 4, unless there is a reset, timeout or error, the major state does not change from 4, and the minor state advances to 1. At the same time, it is inferred that the two boolean flags do not change under any circumstances on this minor cycle:

```
VAL3 minor = 0, VAL4 major = 4
|- BFLAG_ABBR
   (stop,stmplx,bmplx,inst,major,minor,pause,reply,carryused,clkenbp,
    areg,xreg,yreg,preg,rsel,treg,cin,bflag,alucon) =
   bflag

VAL3 minor = 0, VAL4 major = 4
|- STOP_ABBR
   (bflag,stmplx,bmplx,inst,major,minor,pause,reply,carryused,clkenbp,
    areg,xreg,yreg,preg,rsel,treg,cin,alucon,stop) =
   stop
```

The 7-bit control word alucon is sent from the decoder to the ALU, and determines the behaviour of the ALU during its operation. It is computed by DECODE_PERFORM, one of the main sub-functions of the decoder, which takes in the various fields of the instruction and puts out the control values of the DECODER block. The projection function GET_AL selects from the various outputs of DECODE_PERFORM the 7-bit word alucon sent to the ALU.) The alucon register value is inferred (independently of the minor state) to be

```
VAL4 major = 4
|- ALUCON_ABBR(inst,major,bflag,stop) =
   (EL 4(BITS12 inst) => #0011001 |
    GET_AL
    (DECODE_PERFORM
     (WORD4(V(SEG(0,3)(BITS12 inst))),
      WORD3(V(SEG(5,7)(BITS12 inst))),
      WORD2(V(SEG(8,9)(BITS12 inst))),
      WORD2(V(SEG(10,11)(BITS12 inst))),bflag)))
```

That is, there is one 7-bit alucon value for *all* comparison ALU operations (as indicated by bit 4 of the instruction code); while for non-comparisons, DECODE_PERFORM computes the correct control values to be output by the decoder block, and GET_AL selects from these the appropriate alucon value for the ALU operation indicated. Finally, we prove

```
VAL4 major = 4
|- CARRYUSED_ABBR(inst,major,bflag,stop) =
   (EL 4(BITS12 inst) => T |
    ~EL
     5
     (BITS7
      (GET_AL
       (DECODE_PERFORM
        (WORD4(V(SEG(0,3)(BITS12 inst))),
         WORD3(V(SEG(5,7)(BITS12 inst))),
         WORD2(V(SEG(8,9)(BITS12 inst))),
         WORD2(V(SEG(10,11)(BITS12 inst))),bflag)))))
```

meaning that for all comparisons, the carryused value is T; while for non-comparisons it is given by bit 5 of the alucon code.

7.2 Lemmas about Sub-Blocks

The lemmas that are required for proving the **Class 2** theorems exemplified above concern the various functions and sub-functions (such as TIMING) making up the Viper definition, and they depend on no further layers of lemmas, but simply on the original definitions. We call these '**Class 1**' lemmas. To see what is required, consider again the expression for the a-register (page 47). The first set of **Class 1** lemmas required to unfold this expression concern the function TIMING. We prove:

```
VAL3 minor = 0, VAL4 major = 4
|- TIMING(major,minor,pause,reply,carryused) = T,T,F,F,F

VAL3 minor = 1, VAL4 major = 4
|- TIMING(major,minor,pause,reply,carryused) = T,T,F,F,F

VAL3 minor = 2, VAL4 major = 4
|- TIMING(major,minor,pause,reply,carryused) = T,T,F,F,F

VAL3 minor = 3, VAL4 major = 4
|- TIMING(major,minor,pause,reply,carryused) = T,T,~carryused,F,F

VAL3 minor = 4, VAL4 major = 4
|- TIMING(major,minor,pause,reply,carryused) = T,carryused,carryused,F,F

VAL3 minor = 5, VAL4 major = 4
|- TIMING(major,minor,pause,reply,carryused) = T,~carryused,F,F,F

VAL3 minor = 6, VAL4 major = 4
|- TIMING(major,minor,pause,reply,carryused) = T,T,F,F,F

VAL3 minor = 7, VAL4 major = 4
|- TIMING(major,minor,pause,reply,carryused) = T,T,F,F,F
```

These theorems give the complete timing pattern for the major state 4 (PERFORM ALU). They are proved simply by unfolding the definition of TIMING under the various assumptions and simplifying. Along with the definitions of THIRD, FOURTH and FIFTH', the timing theorems enable the AREG_ABBR expressions to be reduced to a form with definite values for the TIMING sub-expressions. For example, assuming respectively that minor is 0, 3 and 4, we infer:

```
VAL3 minor = 0, VAL4 major = 4
|- AREG_ABBR
   (areg,xreg,yreg,preg,rsel,addr,bflag,major,minor,pause,reply,carryused,
    e_data_in,e_resetbar,e_errorbar,e_stepbar,e_reply,treg,cin,alucon,clkenb,inst) =
   FST
   (REGISTERS
    (FST
     (EXTERNAL
      (REGSELECT(areg,xreg,yreg,preg,rsel),addr,bflag,F,major,
       e_data_in,e_resetbar,e_errorbar,e_stepbar,e_reply)),
     FST
     (ALU(REGSELECT(areg,xreg,yreg,preg,rsel),treg,cin,bflag,alucon)),
     clkenb,F,F,areg,xreg,yreg,preg,treg,addr,inst))

VAL3 minor = 3, VAL4 major = 4
|- AREG_ABBR
   (areg,xreg,yreg,preg,rsel,addr,bflag,major,minor,pause,reply,carryused,
    e_data_in,e_resetbar,e_errorbar,e_stepbar,e_reply,treg,cin,alucon,clkenb,inst) =
   FST
   (REGISTERS
    (FST
     (EXTERNAL
      (REGSELECT(areg,xreg,yreg,preg,rsel),addr,bflag,F,major,
       e_data_in,e_resetbar,e_errorbar,e_stepbar,e_reply)),
     FST
     (ALU(REGSELECT(areg,xreg,yreg,preg,rsel),treg,cin,bflag,alucon)),
     clkenb,~carryused,F,areg,xreg,yreg,preg,treg,addr,inst))

VAL3 minor = 4, VAL4 major = 4
|- AREG_ABBR
   (areg,xreg,yreg,preg,rsel,addr,bflag,major,minor,pause,reply,carryused,
    e_data_in,e_resetbar,e_errorbar,e_stepbar,e_reply,treg,cin,alucon,clkenb,inst) =
   FST
   (REGISTERS
    (FST
     (EXTERNAL
      (REGSELECT(areg,xreg,yreg,preg,rsel),addr,bflag,F,major,
       e_data_in,e_resetbar,e_errorbar,e_stepbar,e_reply)),
     FST
     (ALU(REGSELECT(areg,xreg,yreg,preg,rsel),treg,cin,bflag,alucon)),
     clkenb,carryused,F,areg,xreg,yreg,preg,treg,addr,inst))
```

(The other five theorems of the set have the same conclusion as the first one above.) This reveals that we next need some further **Class 1** lemmas about the function REGISTERS. The first one simplifies the definition when the fifth argument to REGISTERS is false – this applies when the minor state is 3 or 4. (The fifth argument, corresponding to the variable '`strobeaddr`' in the original definition, page 33, indicates to DATAREG when the address register,'`addr`', should be loaded.) In that case, new values of the seven registers may be chosen, depending on the decoder's control line `clkenb` and on the timing line `regstrobe`:

```
|- REGISTERS
   (extdatabar,aoutbar,clkenb,regstrobe,F,areg,xreg,yreg,preg,treg,addr,inst) =
   ((~EL 0(BITS7 clkenb) /\ regstrobe) => NOT32 aoutbar | areg),
   ((~EL 2(BITS7 clkenb) /\ regstrobe) => NOT32 aoutbar | xreg),
   ((~EL 3(BITS7 clkenb) /\ regstrobe) => NOT32 aoutbar | yreg),
   ((~EL 1(BITS7 clkenb) /\ regstrobe) =>
    NOT20(WORD20(V(SEG(0,19)(BITS32 aoutbar)))) | preg),
   ((~(~EL 5(BITS7 clkenb) /\ regstrobe)) => treg |
    ((~EL 6(BITS7 clkenb) /\ EL 4(BITS7 clkenb)) => NOT32 extdatabar |
     ((EL 6(BITS7 clkenb) /\ EL 4(BITS7 clkenb)) =>
      WORD32(VAL20(NOT20(WORD20(V(SEG(0,19)(BITS32 aoutbar)))))) |
      ((~EL 6(BITS7 clkenb) /\ ~EL 4(BITS7 clkenb)) =>
       WORD32(V(SEG(0,19)(BITS32(NOT32 extdatabar)))) | ARB)))),
   addr,
   ((~EL 4(BITS7 clkenb) /\ regstrobe) =>
    WORD12(V(SEG(20,31)(BITS32(NOT32 extdatabar)))) | inst)
```

This says that the a-, x-, and y-registers and the program counter may each take on the (negated) value computed by the ALU when it is the indicated destination

register. The temporary register may take on various values because it has various uses in ALU operations. The address-register does not change in this case, and the instruction register might or might not take on the top twelve bits of the data input from memory.

To simplify the **Class 2** theorem for the `minor = 0` case, we prove a theorem which further simplifies the above result by adding the assumption that the fourth input is *also* false. (The fourth argument, corresponding to the variable '`regstrobe`' in the original definition, indicates that the results of certain ALU operations are stable.) In this case none of the seven registers is changed:

```
|- REGISTERS(extdatabar,aoutbar,clkenb,F,F,areg,xreg,yreg,preg,treg,addr,inst) =
   areg,xreg,yreg,preg,treg,addr,inst
```

To prove the first of these two lemmas requires an ML procedure which uses the definitions of REGISTER's sub-functions (CLOCK_REGA and so on) to simplify the definition of REGISTERS under the appropriate assumptions. The second is a simple logical consequence.

Using the lemmas gives us simplified values for the three sample a-register expressions, which in the context of specific values of `clkenb` and `carryused` (during an actual evaluation sequence of the block machine) will give specific a-register values:

```
VAL3 minor = 0, VAL4 major = 4
|- AREG_ABBR
   (areg,xreg,yreg,preg,rsel,addr,bflag,major,minor,pause,reply,
    carryused,e_data_in,e_resetbar,e_errorbar,e_stepbar,e_reply,
    treg,cin,alucon,clkenb,inst) =
   areg

VAL3 minor = 3, VAL4 major = 4
|- AREG_ABBR
   (areg,xreg,yreg,preg,rsel,addr,bflag,major,minor,pause,reply,carryused,
    e_data_in,e_resetbar,e_errorbar,e_stepbar,e_reply,treg,cin,alucon,clkenb,inst) =
   ((~EL 0(BITS7 clkenb) /\ ~carryused) =>
    NOT32
    (FST
     (ALU
      (REGSELECT(areg,xreg,yreg,preg,rsel),treg,cin,bflag,alucon))) |
    areg)

VAL3 minor = 4, VAL4 major = 4
|- AREG_ABBR
   (areg,xreg,yreg,preg,rsel,addr,bflag,major,minor,pause,reply,carryused,
    e_data_in,e_resetbar,e_errorbar,e_stepbar,e_reply,treg,cin,alucon,clkenb,inst) =
   ((~EL 0(BITS7 clkenb) /\ carryused) =>
    NOT32
    (FST
     (ALU
      (REGSELECT(areg,xreg,yreg,preg,rsel),treg,cin,bflag,alucon))) |
    areg)
```

To put the **Class 1** and **Class 2** lemmas mentioned so far into context, it should be obvious that there are a very large number of them. **Class 1** theorems about the timing patterns have to be inferred for *each* of the ten major states at *each* of eight minor states; the REGISTERS theorems have to be proved for true, false and unknown values of the two key arguments; in some cases, lemmas are also required

to simplify the EXTERNAL subexpressions; and analagous theorems have to be proved about *each* of the ten blocks at key specific values. **Class 2** theorems analagous to the theorems about the a-register have to be proved for all of the thirty-one state values and the five outputs, each of these at all combinations of the ten major and eight minor states.

None of the many derivations is difficult, and all can be done easily (and largely automatically) by a set of simple general-purpose ML procedures. The ML procedures required become clear as the unfolding of the block model progresses and facts required become clear. The main point of interest is really the sheer *number* of facts required and the total size of the proofs in time and space (see Section 10).

7.3 The Progression of the Registers

Once the hiererchy of theorems is developed to the point of the **Class 2** lemmas about the register expressions, it is a simple matter to successively substitute into the **Sequencing Theorem** the values for the abbreviation expressions at the eight minor states. By doing this (and using a slightly clever unfolding procedure for recursive unfoldings) a set of eight theorems is proved, all of them of the form sketched below. The theorem for the case `minor_sig n = 0`, for example, with some of the main register values, is shown below[22]:

```
|- (VAL3(minor_sig n) = 0) ==>
   (VAL4(major_sig n) = 4) ==>
   (!n.
     areg_sig(n+1),xreg_sig(n+1),yreg_sig(n+1),preg_sig(n+1),
     treg_sig(n+1),addr_sig(n+1),inst_sig(n+1),
     error_sig(n+1),pause_sig(n+1),reply_sig(n+1),reset_sig(n+1),
     e_iobar_sig n,e_data_out_sig n,e_address_sig n,e_strobebar_sig n,e_writebar_sig n,
     ram_sig(n+1),e_data_in_sig n,
     minor_sig(n+1),major_sig(n+1),bflag_sig(n+1),stop_sig(n+1),
     count_sig(n+1),alucon_sig(n+1),bmplx_sig(n+1),carryused_sig(n+1),
     cin_sig(n+1),clkenb_sig(n+1),clkenbp_sig(n+1),rsel_sig(n+1),stmplx_sig(n+1),
     e_bflag_sig n,e_majorstate_sig n,e_stopped_sig n,e_perform_sig n,e_fetchbar_sig n =
     WHOLE_BLOCK_NEXT
     (areg_sig n,xreg_sig n,yreg_sig n,preg_sig n,treg_sig n,addr_sig n,inst_sig n,
      error_sig n,pause_sig n,reply_sig n,reset_sig n,
      e_iobar_sig n,e_data_out_sig n,e_address_sig n,e_strobebar_sig n,e_writebar_sig n,
      ram_sig n,e_data_in_sig n,
      minor_sig n,major_sig n,bflag_sig n,stop_sig n,count_sig n,alucon_sig n,
      bmplx_sig n,carryused_sig n,cin_sig n,clkenb_sig n,clkenbp_sig n,
      rsel_sig n,stmplx_sig n,
      e_resetbar_sig n,e_errorbar_sig n,e_stepbar_sig n,e_reply_sig n)) ==>

   (areg_sig(n+1) = areg_sig n) /\
       .
       .
       .

   (minor_sig(n+1) =
    ((reset_sig n \/ (count_sig n = #111111) \/ error_sig n) => #000 | #001)) /\

   (major_sig(n+1) =
    (reset_sig n => #0010 |
     (((count_sig n = #111111) \/ error_sig n) => #1000 | #0100))) /\
```

[22]The register value lemmas are instantiated, for this purpose, to signals at times, with n the initial time

```
    .
    .
    .
(bflag_sig(n+1) = bflag_sig n) /\

(stop_sig(n+1) = stop_sig n) /\
    .
    .
    .
(alucon_sig(n+1) =
 (EL 4(BITS12(inst_sig n)) => #0011001 |
  GET_AL
  (DECODE_PERFORM
   (WORD4(V(SEG(0,3)(BITS12(inst_sig n)))),
    WORD3(V(SEG(5,7)(BITS12(inst_sig n)))),
    WORD2(V(SEG(8,9)(BITS12(inst_sig n)))),
    WORD2(V(SEG(10,11)(BITS12(inst_sig n)))),bflag_sig n)))) /\
    .
    .
    .
(carryused_sig(n+1) =
 (EL 4(BITS12(inst_sig n)) => T |
  ~EL
   5
   (BITS7
    (GET_AL
     (DECODE_PERFORM
      (WORD4(V(SEG(0,3)(BITS12(inst_sig n)))),
       WORD3(V(SEG(5,7)(BITS12(inst_sig n)))),
       WORD2(V(SEG(8,9)(BITS12(inst_sig n)))),
       WORD2(V(SEG(10,11)(BITS12(inst_sig n)))),bflag_sig n)))))) /\
    .
    .
    .
```

This asserts that if the minor signal at some time is 0, the a-register signal at the *next* time is unchanged; and the other signals at the next time change or stay the same as shown. As mentioned, the `major` and `minor` values are key factors in determining the progress of the block machine in time. The `areg` and `bflag` values are typical of block state components which are also components of the high level state[23]. The `alucon` and `carryused` values are typical of the block level control values (not reflected in the high level state) which determine the subsequent behaviour of the block model.

By continuing to substitute as above for minor values 1 to 7, the value of each of the registers can be deduced at each of the minor states. That is, we prove a sequence of theorems analogous to the one above for all eight possible values of `minor`. These contain, for example, the sequence of values of the a-register. Inspection reveals that the a-register shows no change except when the minor state is 3 or 4. This follows from the register value lemmas concerning the a-register (simplified as shown). The fourth theorem in the new sequence tells us that

[23]The block level stop flag is not precisely the same as the high level stop flag. See remarks in the Conclusions section.

```
|- (VAL3(minor_sig n) = 3) ==>
   (VAL4(major_sig n) = 4) ==>
       .
       .
       .
   (areg_sig(n+1) =
    ((~EL 0(BITS7(clkenb_sig n)) /\ ~carryused_sig n) =>
     NOT32
     (FST
      (ALU
       (REGSELECT
        (areg_sig n,xreg_sig n,yreg_sig n,preg_sig n,rsel_sig n),
        treg_sig n,cin_sig n,bflag_sig n,alucon_sig n))) |
     areg_sig n)) /\ ...
```

and the fifth tells us

```
|- (VAL3(minor_sig n) = 4) ==>
   (VAL4(major_sig n) = 4) ==>
       .
       .
       .
   (areg_sig(n+1) =
    ((~EL 0(BITS7(clkenb_sig n)) /\ carryused_sig n) =>
     NOT32
     (FST
      (ALU
       (REGSELECT
        (areg_sig n,xreg_sig n,yreg_sig n,preg_sig n,rsel_sig n),
        treg_sig n,cin_sig n,bflag_sig n,alucon_sig n))) |
     areg_sig n)) /\ ...
```

and the a-register does not change subsequently. (The pattern for the x-, y-, p- and t-registers is similar.) Meanwhile (by analogous inferences based on analogous lemmas), the successive values of the `major` and `minor` registers at each of the minor states are as follows:

```
|- (VAL3(minor_sig n) = 0) ==> (VAL4(major_sig n) = 4) ==>
       .
       .
       .
   (minor_sig(n+1) =
    ((reset_sig n \/ (count_sig n = #111111) \/ error_sig n) => #000 | #001)) /\
   (major_sig(n+1) =
    (reset_sig n => #0010 | (((count_sig n = #111111) \/ error_sig n) => #1000 | #0100))) /\ ...

|- (VAL3(minor_sig n) = 1) ==> (VAL4(major_sig n) = 4) ==>
       .
       .
       .
   (minor_sig(n+1) =
    ((reset_sig n \/ (count_sig n = #111111) \/ error_sig n) => #000 | #010)) /\
   (major_sig(n+1) =
    (reset_sig n => #0010 | (((count_sig n = #111111) \/ error_sig n) => #1000 | #0100))) /\ ...

|- (VAL3(minor_sig n) = 2) ==> (VAL4(major_sig n) = 4) ==>
       .
       .
       .
   (minor_sig(n+1) =
    ((reset_sig n \/ (count_sig n = #111111) \/ error_sig n) => #000 | #011)) /\
   (major_sig(n+1) =
    (reset_sig n => #0010 | (((count_sig n = #111111) \/ error_sig n) => #1000 | #0100))) /\ ...

|- (VAL3(minor_sig n) = 3) ==> (VAL4(major_sig n) = 4) ==>
       .
       .
       .
   (minor_sig(n+1) =
    ((reset_sig n \/ (count_sig n = #111111) \/ error_sig n) => #000 | #100)) /\
   (major_sig(n+1) =
    (reset_sig n =>  #0010 | (((count_sig n = #111111) \/ error_sig n) => #1000 | #0100))) /\ ...
```

```
|- (VAL3(minor_sig n) = 4) ==> (VAL4(major_sig n) = 4) ==>
      .
      .
      .
   (minor_sig(n+1) =
    ((reset_sig n \/ ((count_sig n = #111111) \/ error_sig n) \/
     ~carryused_sig n) => #000 | #101)) /\
   (major_sig(n+1) =
    (reset_sig n => #0010 |
     (((count_sig n = #111111) \/ error_sig n) => #1000 |
      ((~carryused_sig n) => (stop_sig n => #1000 | #0001) | #0100)))) /\ ...

|- (VAL3(minor_sig n) = 5) ==> (VAL4(major_sig n) = 4) ==>
      .
      .
      .
   (minor_sig(n+1) =
    ((reset_sig n \/ ((count_sig n = #111111) \/ error_sig n) \/
      carryused_sig n) => #000 | #110)) /\
   (major_sig(n+1) =
    (reset_sig n => #0010 |
     (((count_sig n = #111111) \/ error_sig n) => #1000 |
      (carryused_sig n => (stop_sig n => #1000 | #0001) | #0100)))) /\ ...

|- (VAL3(minor_sig n) = 6) ==> (VAL4(major_sig n) = 4) ==>
      .
      .
      .
   (minor_sig(n+1) =
    ((reset_sig n \/ (count_sig n = #111111) \/ error_sig n) => #000 | #111)) /\
   (major_sig(n+1) =
    (reset_sig n => #0010 | (((count_sig n = #111111) \/ error_sig n) => #1000 | #0100))) /\ ...

|- (VAL3(minor_sig n) = 7) ==> (VAL4(major_sig n) = 4) ==>
      .
      .
      .
  (minor_sig(n+1) = #000) /\
  (major_sig(n+1) =
    (reset_sig n => #0010 | (((count_sig n = #111111) \/ error_sig n) => #1000 | #0100))) /\ ...
```

As can be seen, the advancement of `major` and `minor` depends on the absence of signals for stopping, resetting or timing-out the machine.

The two boolean flags change when the minor state is 3 and again when it is 4 (and at no other minor states). For the b-flag value we have

```
|- (VAL3(minor_sig n) = 3) ==> (VAL4(major_sig n) = 4) ==>
       .
       .
       .
   (bflag_sig(n+1) =
    SND
    (BANDSTOP
     (stmplx_sig n,bmplx_sig n,FSELECT(inst_sig n),~carryused_sig n,clkenbp_sig n,
      SND
      (ALU
       (REGSELECT
        (areg_sig n,xreg_sig n,yreg_sig n,preg_sig n,rsel_sig n),
       treg_sig n,cin_sig n,bflag_sig n,alucon_sig n)),stop_sig n,bflag_sig n))) /\ ...

|- (VAL3(minor_sig n) = 4) ==> (VAL4(major_sig n) = 4) ==>
       .
       .
       .
   (bflag_sig(n+1) =
    SND
    (BANDSTOP
     (stmplx_sig n,bmplx_sig n,FSELECT(inst_sig n),carryused_sig n,clkenbp_sig n,
      SND
      (ALU
       (REGSELECT
        (areg_sig n,xreg_sig n,yreg_sig n,preg_sig n,rsel_sig n),
        treg_sig n,cin_sig n,bflag_sig n,alucon_sig n)),stop_sig n,bflag_sig n))) /\ ...
```

The bflag values are similar but depend obviously on the second BANDSTOP output rather than the first. Both stop and bflag also depend on the second ALU output – the 9-bit word (conditions, in Figure 1) which codes the conditions on the source register and on the computed value needed to check for errors.

The carryused and alucon signals do not depend on the minor state, just the major state, so they do not change throughout the transitions.

7.4 The Individual Minor State Transitions

In order to compose the minor state transitions to yield their *cumulative* effects, certain assumptions must be made. First, in order to advance the major and minor states (i.e. to resolve their expressions to particular values), some assumptions must be made about certain *internal* lines of the Viper block model at the starting time (time n). For example, it is necessary to assume that

```
(reset_sig n = F) /\
(error_sig n = F) /\
((count_sig n = #111111) = F)
```

That is, the block model cannot be 'run' (starting with a minor state 0 and a major state 4) if the reset or error value is initially set, or if the timout-counter is initially at its maximum value. Of these three, only the count signal is genuinely internal to the block model. The reset and error values stem from external inputs (though, as mentioned earlier, this is not deducible from Figure 1 or any definitions), so that at every time after time n, these two can be expressed in terms of external values at the previous time. In particular, as the transitions are composed, reset and error take on the values ~e_resetbar and ~e_errorbar respectively[24]. To propagate the correct chain of subsequent values for e_resetbar and e_errorbar, *their* initial values must also be assumed[25]:

```
(e_resetbar_sig n = T) /\
(e_errorbar_sig n = T)
```

To evaluate the destination register expressions (e.g. the a-register expression on the minor cycles in which it *does* change), it is necessary to unfold the function ALU. The definition has the form

[24]The fact that these connections can be deduced is the reason that they do not have to be indicated in the formal specifications of the blocks, nor drawn in Figure 1.

[25]This is consistent with the RSRE informal explanation (Annex A) of [22] in which these two values are "normally held high".

```
|- ALU(rbar,treg,cin,bflag,alucon) =
   (    .
        .
        .
    let a_c = BITOP(...,treg,bflag,...,cin,alucon) in
    let aout = GET_AOUT a_c in
    let aoutbar = NOT32 aout in
        .
        .
        .
    aoutbar,conditions)
```

BITOP uses the 7-bit control word alucon (from the decoder), whose thirteen (as it happens) permitted values characterize the operation to be performed. BITOP returns the 32-bit computed value (negated), and a boolean value (used for various purposes); GET_AOUT returns the computed value. (See the Appendix for the full definitions.) The value in the alucon register remains the same throughout the minor state sequence – as we have seen:

```
(EL 4(BITS12(inst_sig n)) => #0011001 |
 GET_AL
 (DECODE_PERFORM
  (WORD4(V(SEG(0,3)(BITS12(inst_sig n)))),
   WORD3(V(SEG(5,7)(BITS12(inst_sig n)))),
   WORD2(V(SEG(8,9)(BITS12(inst_sig n)))),
   WORD2(V(SEG(10,11)(BITS12(inst_sig n)))),bflag_sig n))))
```

To resolve the alucon expression, EL 4(BITS12(inst_sig n)) (the fifth bit of the initial instruction register value) must be fixed. That is, it must be known whether the instruction indicates a comparison operation, in which case the alucon value #0011001 happens to be the appropriate code; or whether it is not a comparison, in which case the appropriate code signal is computed by the function DECODE_PERFORM of the decoder. That in turn depends on knowing at least the 4-bit function code of ther current instruction, i.e. the value of WORD4(V(SEG(0,3)(BITS12(inst_sig n)))).

Suppose, for example, that the instruction is *not* a comparison, and that the function field indicates addition with detection of overflows. According to the RSRE informal explanations in [22], that case would be indicated by the following assumptions on the current instruction:

```
(EL 4(BITS12(inst_sig n)) = F) /\
(WORD4(V(SEG(0,3)(BITS12(inst_sig n)))) = #0101)
```

For this case, a simple **Class 1** lemma is proved (by the usual unfolding of definitions – see the Appendix for details of DECODE_PERFORM) stating that

```
|- DECODE_PERFORM(#0101,dsf,msf,rsf,bflag) = ...,...,...,F,#0001001,DSFPRIM dsf
```

where DSFPRIM is a sub-function of the decoder which returns (another) 7-bit code, characterizing the destination register. The ALU signal in this case (picked out by the function GET_AL) is #0001001 throughout the sequence. The assumptions also obviously fix the carryused register throughout the sequence to be T (it is fixed by the destination code). Finally, for similar reasons, we also know that

```
cin_sig(n+1) = F /\
clkenb_sig(n+1) = DSFPRIM(WORD3(V(SEG(5,7)(BITS12(inst_sig n)))))
```

throughout, where `clkenb` controls the loading of registers, and `cin` is used for carries and shifts, according to context.

With a specific `alucon` signal (and the others) and the five assumptions about the initial state, it is now possible to deduce a simpler sequence of theorems describing the minor state transitions. For example:

```
|- (VAL3(minor_sig n) = 0) ==>
   (VAL4(major_sig n) = 4) ==>
   (WORD4(V(SEG(0,3)(BITS12(inst_sig n)))) = #0101) ==>
   (EL 4(BITS12(inst_sig n)) = F) ==>
   (e_resetbar_sig n = T) ==>
   (e_errorbar_sig n = T) ==>
   (error_sig n = F) ==>
   (reset_sig n = F) ==>
   ((count_sig n = #111111) = F) ==>
        .
        .
        .
   (minor_sig(n+1) = #001) /\
   (major_sig(n+1) = #0100) /\
        .
        .
        .
```

The subsequent values of the `minor` and `major` registers through the sequence are shown below.

```
(minor_sig(n+1) = #010) /\
(major_sig(n+1) = #0100) /\

(minor_sig(n+1) = #011) /\
(major_sig(n+1) = #0100) /\

(minor_sig(n+1) = #100) /\
(major_sig(n+1) = #0100) /\

(minor_sig(n+1) = #101) /\
(major_sig(n+1) = #0100) /\

(minor_sig(n+1) = #000) /\
(major_sig(n+1) = (stop_sig n => #1000 | #0001)) /\

(minor_sig(n+1) = #111) /\
(major_sig(n+1) = #0100) /\

(minor_sig(n+1) = #000) /\
(major_sig(n+1) = #0100) /\
```

The minor state will therefore advance by increments of 1 until the sixth cycle, at which time it is comes round to 0 again, ready to begin another major cycle. The major state, meanwhile, will stay fixed at 4 until the same sixth cycle, at which time it will then (normally) procede to 1 (FETCH, the beginning of a new fetch-decode-execute loop), unless the stopping flag forces it to 8, the stopped state. (The final two theorems therefore turn out to be superfluous.)

If the minor states had *not* progressed in increments of one, returing to 0 again at some point at or before the last transition, or if the major state had not changed

to 1 (FETCH) or 8 (STOP) at the same point, as intended in the informal specification in [22], then an error would have been indicated – either in the block design of Viper *or* in the HOL representation of the block design. The source of the error would have had to have been traced back heuristically through the analysis, possibly to the design. All that would be certain were that (modulo the correctness of the implementation of HOL) the error could not lie in the proof; see Section 4 for a discussion of the security of HOL proofs. In fact, all of the transitions do work correctly in the Viper block machine.

In conclusion, a full account of the individual transitions has now been inferred (illustrated for the seven typical registers). With the pattern of minor and major sequences established, we are finally in a position to compose together the individual minor state transitions.

7.5 Composing the Minor State Transitions

The sequence of theorems describing the individual minor state transitions through the major state 4 are now used to infer the *cumulative* effects on the various registers of the sequence of minor state transitions. The initial effects are described by the simplified theorem for the minor state 0 (and major state 4)[26]:

```
|- (VAL3(minor_sig n) = 0) ==>
   (VAL4(major_sig n) = 4) ==>
   (WORD4(V(SEG(0,3)(BITS12(inst_sig n)))) = #0101) ==>
   (EL 4(BITS12(inst_sig n)) = F) ==>
   (e_resetbar_sig n = T) ==>
   (e_errorbar_sig n = T) ==>
   (error_sig n = F) ==>
   (reset_sig n = F) ==>
   ((count_sig n = #111111) = F) ==>

   (areg_sig(n+1) = areg_sig n) /\
   (minor_sig(n+1) = #001) /\
   (major_sig(n+1) = #0100) /\
   (bflag_sig(n+1) = bflag_sig n) /\
   (stop_sig(n+1) = stop_sig n) /\
   (alucon_sig(n+1) = #0001001) /\
   (carryused_sig(n+1) = T) /\
   (clkenb_sig(n+1) = DSFPRIM(WORD3(V(SEG(5,7)(BITS12(inst_sig n))))) /\
   (cin_sig(n+1) = F) /\
     .
     .
     .
```

At each subsequent stage until the minor state first returns to 0, the *next* cumulative result in the sequence is inferred from the *current* cumulative theorem. The inference procedure is recursive:

[26] In this respect it would be impossible to do an analysis of the block model without some indication that the minor state 0 was distinguished in this way. This indication is in RSRE's English description rather than in the specification itself.

1. For whatever value from 1 to 7 the minor state of the current cumulative theorem is, the appropriate minor state transition theorem is selected, giving time n+1 values in terms of time n values, for each n. For example, when the minor state is 4, the corresponding state transition theorem tells us the next states in terms of the current – for example:

```
(areg_sig(n+1) =
 ((~EL 0(BITS7(clkenb_sig n)) /\ carryused_sig n) =>
  NOT32
  (FST
   (ALU
    (REGSELECT
     (areg_sig n,xreg_sig n,yreg_sig n,preg_sig n,rsel_sig n),
     treg_sig n,cin_sig n,bflag_sig n,alucon_sig n))) |
  areg_sig n)) /\

(minor_sig(n+1) = ((~carryused_sig n) => #000 | #101))
```

   At this stage (i.e. time n+4) the current *cumulative* theorem so far happens to assert that

```
(areg_sig(n+4) = areg_sig n) /\
(inst_sig(n+4) = inst_sig n) /\
(minor_sig(n+4) = #100) /\
(major_sig(n+4) = #0100) /\
(bflag_sig(n+4) = bflag_sig n) /\
(stop_sig(n+4) = stop_sig n) /\
(carryused_sig(n+4) = T) /\
(clkenb_sig(n+4) = DSFPRIM(WORD3(V(SEG(5,7)(BITS12(inst_sig n))))) /\
(cin_sig(n+4) = F) /\
(rsel_sig(n+4) = WORD2(V(SEG(10,11)(BITS12(inst_sig n))))) /\
(bmplx_sig(n+4) = #000) /\
(clkenbp_sig(n+4) = ~EL 1(BITS7(DSFPRIM(WORD3(V(SEG(5,7)(BITS12(inst_sig n)))))))) /\
(stmplx_sig(n+4) = #011) /\
(alucon_sig(n+4) = #0001001) /\ ...
```

   It gives the time-n+4 values in terms of the initial values.

2. The appropriate transition theorem is then instantiated to the time of the current theorem, giving a new transition theorem. In the example n is instantiated to n+4, giving a transition theorem useful in the present circumstances: it expresses the time n+5 values in terms of the known time n+4 values. Among other things, the new transition theorem asserts that

```
(areg_sig(n+5) =
 ((~EL 0(BITS7(clkenb_sig(n+4))) /\ carryused_sig(n+4)) =>
  NOT32
  (FST
   (ALU
    (REGSELECT
     (areg_sig(n+4),xreg_sig(n+4),yreg_sig(n+4),preg_sig(n+4),rsel_sig(n+4)),
     treg_sig(n+4),cin_sig(n+4),bflag_sig(n+4),alucon_sig(n+4)))) |
  areg_sig(n+4))) /\

(minor_sig(n+5) = ((~carryused_sig(n+4)) => #000 | #101)) /\

(major_sig(n+5) =
 ((~carryused_sig(n+4)) => (stop_sig(n+4) => #1000 | #0001) | #0100)) /\
```

```
(stop_sig(n+5) =
 FST
 (BANDSTOP
  (stmplx_sig(n+4),bmplx_sig(n+4),FSELECT(inst_sig(n+4)),carryused_sig(n+4),
   clkenbp_sig(n+4),
   SND
   (ALU
    (REGSELECT
     (areg_sig(n+4),xreg_sig(n+4),yreg_sig(n+4),preg_sig(n+4),rsel_sig(n+4)),
     treg_sig(n+4),cin_sig(n+4),bflag_sig(n+4),alucon_sig(n+4))),
   stop_sig(n+4),bflag_sig(n+4)))) /\ ...
```

3. The last step consists in substituting the known values given by the cumulative theorem so far into the new transition theorem. In the example, the known time n+4 values are substituted for the time-n+4 terms (this step is justified by the rule of substitution of equals for equals.) The substitution puts the new cumulative theorem entirely in terms of the time-n values. Thus, for example, the new cumulative theorem asserts:

```
(areg_sig(n+5) =
 ((~EL 0(BITS7(DSFPRIM(WORD3(V(SEG(5,7)(BITS12(inst_sig n)))))))) =>
  NOT32
  (FST
   (ALU
    (REGSELECT
     (areg_sig n,xreg_sig n,yreg_sig n,preg_sig n,WORD2(V(SEG(10,11)(BITS12(inst_sig n))))),
     treg_sig n,F,bflag_sig n,#0001001))) |
  areg_sig n)) /\

(minor_sig(n+5) = #101) /\

(major_sig(n+5) = #0100) /\

(stop_sig(n+5) =
 FST
 (BANDSTOP
  (#011,#000,FSELECT(inst_sig n),T,
   ~EL 1(BITS7(DSFPRIM(WORD3(V(SEG(5,7)(BITS12(inst_sig n))))))),
   SND
   (ALU
    (REGSELECT
     (areg_sig n,xreg_sig n,yreg_sig n,preg_sig n,WORD2(V(SEG(10,11)(BITS12(inst_sig n))))),
     treg_sig n,F,bflag_sig n,#0001001)),
   stop_sig n,bflag_sig n)))
```

so that now the a-register, if it is the desination register indicated, takes on the 32-bit result computed by the ALU. The stopping value on this cycle depends on the other computed result – the 9-bit word encoding the knowledge about error conditions. (In this case the stopping expression will evaluate to T if an addition overflow has occurred.)

Here, a new set of **Class 2** lemmas comes into play; the a-register expression, for example, requires a lemma giving the actual computed value (i.e. the first of the two outputs) of the ALU on the values shown. That lemma in fact implies that

```
areg_sig(n+5) =
((~EL 0(BITS7(DSFPRIM(WORD3(V(SEG(5,7)(BITS12(inst_sig n)))))))) =>
 WORD32
 (V
  (SEG
   (0,31)
   (BITS33
    (WORD33
     ((VAL32
       (NOT32
        (REGSELECT
         (areg_sig n,xreg_sig n,yreg_sig n,preg_sig n,
          WORD2(V(SEG(10,11)(BITS12(inst_sig n)))))))) +
      (VAL32(treg_sig n)))))) |
 areg_sig n)
```

Likewise, lemmas for the stopping expression imply that[27]

```
stop_sig(n+5) =
~EL 1(BITS7(DSFPRIM(WORD3(V(SEG(5,7)(BITS12(inst_sig n)))))))) /\
~(V
  (SEG
   (20,31)
   (BITS32
    (WORD32
     (V
      (SEG
       (0,31)
       (BITS33
        (WORD33
         ((VAL32
           (NOT32
            (REGSELECT
             (areg_sig n,xreg_sig n,yreg_sig n,preg_sig n,
              WORD2(V(SEG(10,11)(BITS12(inst_sig n)))))))) +
          (VAL32(treg_sig n))))))))) =
  0) \/

(EL
 31
 (BITS32
  (NOT32
   (REGSELECT
    (areg_sig n,xreg_sig n,yreg_sig n,preg_sig n,
     WORD2(V(SEG(10,11)(BITS12(inst_sig n)))))))) =
 EL 31(BITS32(treg_sig n))) /\
~(EL
  31
  (BITS32
   (NOT32
    (REGSELECT
     (areg_sig n,xreg_sig n,yreg_sig n,preg_sig n,
      WORD2(V(SEG(10,11)(BITS12(inst_sig n)))))))) =
  EL
  31
  (BITS32
   (WORD32
    (V
     (SEG
      (0,31)
      (BITS33
       (WORD33
        ((VAL32
          (NOT32
           (REGSELECT
            (areg_sig n,xreg_sig n,yreg_sig n,preg_sig n,
             WORD2(V(SEG(10,11)(BITS12(inst_sig n)))))))) +
         (VAL32(treg_sig n)))))))))
```

The a-register result means that if the a-register is the intended destination, then the value initially in the temporary register is added to the (inverse of) the value

[27]The accumulation of values produces a rather large term – it could be subsequently abbreviated if desired.

initially in the source (a-) register to give a 33-bit sum, of which the thirty-third bit is simply ignored (by taking the 32-bit lower segment). If the a-register is not the intended destination it remains unchanged from its initial value.

The stopping value will be true if either of two conditions hold. The first disjunct tests whether the intended destination is the (20-bit) program counter in the case that the sum to be stored there actually uses more than twenty bits. The second disjunct compares the top bit of the source register to the top bits of both the temporary register and the sum, in order to detect addition overflow[28].

At time `n+6`, the minor state becomes 0 again, ready to start a new cycle; and neither the a-register nor the stop flag change. The major state changes in the last cycle from 4, which it has been thus far, to either 8 (STOP) or 1 (FETCH), depending on the stop flag. In this way the final cumulative theorem represents the processing of the single major state 4 (PERFORM ALU) and specifies the *next* major state to be processed.

In the instantiation phase of the recursive accumulation procedure, new assumptions may be generated. In the example, the old cumulative theorem's assumptions included

```
(error_sig n = F) /\
((count_sig n = #111111) = F) /\
(reset_sig n = F) /\

(e_resetbar_sig n = T) /\
(e_resetbar_sig(n+1) = T) /\
(e_resetbar_sig(n+2) = T) /\
(e_resetbar_sig(n+3) = T) /\

(e_errorbar_sig n = T) /\
(e_errorbar_sig(n+1) = T) /\
(e_errorbar_sig(n+2) = T) /\
(e_errorbar_sig(n+3) = T)
```

while the new cumulative theorem's assumptions add to those

```
(e_resetbar_sig(n+4) = T) /\
(e_errorbar_sig(n+4) = T)
```

In the *final* cumulative theorem, `e_errorbar` and `e_resetbar` are assumed to hold steady up to time `n+5`.

The cycle of steps which produces the final cumulative theorem is implemented in HOL by a recursive ML procedure. The procedure produces the next cumulative theorem at each recursive call, while at the same time monitoring the progress of the minor state. The first time after the start that the minor value reaches 0 again, the procedure stops and produces the current cumulative theorem as its final result. In this way, the entire chain of inferences culminating in the final cumulative result

[28]This takes some thinking about.

is produced automatically – and yet fully formally (once the lemmas are proved). This is an illustration of the power of the linked programming language and the logic in LCF-type systems.

7.6 Lemmas for the Composed Transitions

Stepping experimentally through the procedure, it becomes clear what **Class 2** (and hence **Class 1**) lemmas are required en route. These lemmas make it clearer how the sum and overflow check evolve in the chain of cumulative results. For example, inferring the time-n+5 value of the a-register shown earlier requires a lemma about the function ALU (on specific values). Namely, we need to know the first output (the computed value) of ALU for the add case:

```
|- FST
   (ALU
    (REGSELECT(areg,xreg,yreg,preg,WORD2(V(SEG(10,11)(BITS12 inst)))),treg,F,bflag,#0001001)) =
   NOT32
   (WORD32
    (V
     (SEG
      (0,31)
      (BITS33
       (WORD33
        ((VAL32
          (NOT32
           (REGSELECT
            (areg,xreg,yreg,preg,WORD2(V(SEG(10,11)(BITS12 inst)))))) +
         (VAL32 treg)))))))
```

Clearly, this specifies the sum of the selected register and the temporary register. It is proved as usual in a forward manner, by unfolding and simplifying, and using various further **Class 1** lemmas.

Similarly, two lemmas produce the time-n+5 stopping value. The first concerns the *second* output of the application of the function ALU to some specific values. The lemma specifies the 9-bit error conditions code for addition:

```
|- SND
   (ALU
    (REGSELECT(areg,xreg,yreg,preg,WORD2(V(SEG(10,11)(BITS12 inst)))),
     treg,F,bflag,#0001001)) =
   WORD9
   (V
    [EL
     31
     (BITS32
      (WORD32
       (V
        (SEG
         (0,31)
         (BITS33
          (WORD33
           ((VAL32
             (NOT32
              (REGSELECT
               (areg,xreg,yreg,preg,WORD2(V(SEG(10,11)(BITS12 inst)))))) +
            (VAL32 treg)))))));
     ...;
     ...;
```

```
EL
31
(BITS32
 (NOT32
  (REGSELECT
   (areg,xreg,yreg,preg,WORD2(V(SEG(10,11)(BITS12 inst)))))));
EL 31(BITS32 treg);
...;
~(V
  (SEG
   (20,31)
   (BITS32
    (WORD32
     (V
      (SEG
       (0,31)
       (BITS33
        (WORD33
         ((VAL32
           (NOT32
            (REGSELECT
             (areg,xreg,yreg,preg,WORD2(V(SEG(10,11)(BITS12 inst)))))) +
          (VAL32 treg)))))))) =
  0);
...;
...])
```

In the 9-bit word produced, bit 8 is the top bit of the sum; bit 5 is the top bit of the source register; bit 4 is the top bit of the temporary register; and bit 2 indicates whether the top 12 bits of the sum are used (it is false if they are).

The second lemma concerns the first output of the function BANDSTOP when applied to certain values; this gives the stopping value:

```
|- FST
   (BANDSTOP
    (#011,#000,fsf,T,clkenbp,
     SND
     (ALU
      (REGSELECT(areg,xreg,yreg,preg,WORD2(V(SEG(10,11)(BITS12 inst)))),
       treg,F,bflag,#0001001)),stop,bflag)) =
   clkenbp /\
   EL 2(BITS9
        (SND
         (ALU
          (REGSELECT(areg,xreg,yreg,preg,WORD2(V(SEG(10,11)(BITS12 inst)))),
           treg,F,bflag,#0001001)))) \/

   (EL 5(BITS9
         (SND
          (ALU
           (REGSELECT(areg,xreg,yreg,preg,WORD2(V(SEG(10,11)(BITS12 inst)))),
            treg,F,bflag,#0001001)))) =
    EL 4(BITS9
         (SND
          (ALU
           (REGSELECT(areg,xreg,yreg,preg,WORD2(V(SEG(10,11)(BITS12 inst)))),
            treg,F,bflag,#0001001))))) /\
   ~(EL 5(BITS9
          (SND
           (ALU
            (REGSELECT
             (areg,xreg,yreg,preg,WORD2(V(SEG(10,11)(BITS12 inst)))),treg,F,bflag,#0001001)))) =
     EL 8(BITS9
          (SND
           (ALU
            (REGSELECT
             (areg,xreg,yreg,preg,WORD2(V(SEG(10,11)(BITS12 inst)))),treg,F,bflag,#0001001)))))
```

The first disjunct codes the information that the intended destination of the computed result is the program counter (`clkenb` indicates that) *and* that the top twelve

bits of the computed 32-bit ALU result are used. This means that the machine must stop, since the computed result will not fit into the 20-bit p-register. The second disjunct is the overflow condition for addition, which if true must also cause the machine must also stop.

These **Class 2** lemmas depend on still simpler **Class 1** lemmas about the functions ALU and BANDSTOP, on specific values. They follow from the basic Viper block definitions by subsitution of the specific values (i.e. unfolding and simplifying again). There are naturally many more lemmas required of a similar form, for other possible ALU operations, etc. Although all involve different sets of control signals and so on, the ML procedures that generate the proofs are fairly uniform. The required lemmas are all identified as illustrated – by interactively deriving the minor state transitions (in sequence one-by-one), accumulating results, and analyzing the resulting expressions for sub-expressions whose evaluation is essential for producing the *next* theorem in the sequence. This process requires human intelligence to identify the relevant sub-expressions, as well as machine assistance to carry out the massive inferences. In theory, more of the process could be automated, but that is a research problem of its own, on the boundaries of artificial intelligence.

7.7 Remarks

It is worth stressing at this point that *all* of the proofs in this section are produced by forward reasoning, i.e. straightforward unfoldings of the basic block model definitions or previous lemmas. None of the results were planned or foreseen; we have simply deduced some consequences of the block representation function we derived, and hence analyzed it. The results can usefully be compared with the intended results, or just checked for plausibility; but in any case, the intentions were in no way taken into account in the analysis. The only prior knowledge required for the analysis thus far was the fact that minor cycles always begin at 0.

The minor state analysis proceeds in a similar way for each of the other arithmetic-logic major states, and for all of the non-ALU major states. There are thirty-two ALU operations in all: sixteen comparisons, four arithmetic operations, five logical operations, two read operations, four shifts, and finally, procedure calls. (For purposes of analysis, forty-one ALU operations are considered later on in the analysis, because breaking some up into cases depending on whether the p-register is the destination simplifies matters.) There are nine non-ALU major states, as listed in Section 5.

It is also worth emphasizing the large total size of the cumulative-result theorems; for example, the one used in the example was nine pages long (when pretty-printed in full), and it is only one average-sized theorem among many similar ones. These depend on very large numbers of lemmas of the type illustrated.

Briefly, as this is needed later, the (six) minor transitions comprising the FETCH operation produce a final cumulative theorem asserting that in fetching an instruction, the program counter is (in a roundabout way) incremented:

```
(preg_sig(n+6) =
 NOT20
 (WORD20
  (V
   (SEG
    (0,19)
    (BITS32
     (NOT32
      (WORD32
       (V
        (SEG
         (0,31)
         (BITS33(WORD33((VAL32(WORD32(VAL20(preg_sig n)))) + 1)))))))))))))
```

It also tells us that the temporary register takes on the address field of the new, fetched instruction:

```
(treg_sig(n+6) =
 WORD32
 (V
  (SEG
   (0,19)
   (BITS32
    (FETCH21(ram_sig n)(WORD21(V(CONS F(BITS20(preg_sig n))))))))))
```

Likewise, the `inst` signal in the end holds the (12-bit) instruction code of the new instruction. The `stop` signal becomes true *only* if incrementing the program counter makes it more than twenty significant bits long. The `major` signal can hold a wide variety of values, determined by analysis of the fields of the new fetched instruction. This makes sense, since any other of the major states may follow a fetch operation. The conditional expression representing this analysis is quite complex because of the number of combinations possible. The a-register (nor the others, aside from the program counter) does not change.

The assumptions generated in the process of accumulating minor state effects through the FETCH node are: the external signals `e_errorbar`, `e_resetbar`, `e_reply` and `e_stepbar` are true from time `n` to time `n+5`. Also, the internal signals `reset`, `error` and `pause` are false at time `n`, while `reply` is true at time `n`, and `count` is not 63 at time `n`.

The number of minor transitions required varies over the major states. They range in length from one cycle (STOP) to three cycles (READMEM) to four cycles (PRECALL, RESET, WRITEMEM, WRITEIO and READIO) to five cycles

(PERFORM ALU operations involving right shifts, calls, reads, and logic operations) to six cycles (FETCH, INDEX, and PERFORM ALU operations involving left shifts, adds, subtracts and comparisons).

To summarize, we have shown through a process of inference how the block definitions (including the definition of the minor state block) together with the information in Figure 1, and the knowledge that minor cycles start at minor state 0, determine the minor state transitions which comprise the major states of the Viper block machine. The minor state transitions give information about the ultimate values on the thirty-one lines of the block model (including the line indicating the next major state), as well as the values on the five lines to the outside world. These values reflect the block state resulting from the execution of each of the major states, such as fetching an instruction or performing an addition operation.

The values determined for the major line in particular put us finally in a position to analyze the *major* state transitions of the block model.

8 Using the Representation: The Major State Transitions

In Section 7 we have shown how the logical analysis of the minor state transitions is carried out. A bonus of that analysis is the information on how the major states follow each other; each minor state transition through a particular major state yields an expression for selecting the next major state. Given adequate case-assumptions to resolve these expressions into definite values, sequences of major states can be composed in the same way that sequences of minor states were composed. This is a much simpler process than composing the minor states; for one thing, there are fewer lemmas needed. The first problem, therefore, is to work out what case assumptions are required.

8.1 The Major State Transition Conditions

In Section 7 we briefly sketched the effects of executing a FETCH instruction: a fresh instruction is produced from the memory according to the program counter and its fields are placed in the appropriate registers of the block machine. One of the results of the FETCH operation, appearing in the `major` register, is a long, complex expression specifying the next major state. The expression is a conditional in which *any* of the ten major states (including FETCH itself, if the instruction in question is a no-op) can follow the FETCH state. The conditional expression can

be helpfully shortened by introducing an abbreviation (as introduced in Section 4):

```
|- FETCH_ABBR(ram,preg) =
   BITS12
   (WORD12
    (V
     (SEG(20,31)(BITS32(FETCH21 ram(WORD21(V(CONS F(BITS20 preg))))))))
```

which denotes the list of booleans representing the twelve bits of the instruction code of the new instruction.

In selecting the next state after FETCH, the conditional expression branches according to the various fields of the 12-bit instruction. It will choose state 4 (PERFORM ALU), for example, under several different sets of circumstances, corresponding to the several possible ALU operations. To characterize the choice of major state 4, for example, we define a predicate (called c4) which holds (over certain components of the visible state) if and only if PERFORM ALU is the major state following the FETCH. This predicate is constructed by analyzing the conditional expression (derived from the block specifications) for changing major states; that is, it follows *entirely* from the block definitions, and is neither invented for the purpose nor derived from the documentation of Viper. The predicate works out as follows[29]:

```
|- c4(ram,preg,areg,xreg,yreg,bflag) =
   (V
    (SEG
     (20,31)
     (BITS32
      (WORD32
       (V(SEG(0,31)(BITS33(WORD33((VAL32(WORD32(VAL20 preg))) + 1)))))))) =
    0) /\

   (EL 4(FETCH_ABBR(ram,preg)) /\
    (VAL2(WORD2(V(SEG(8,9)(FETCH_ABBR(ram,preg))))) = 0) \/

    ~EL 4(FETCH_ABBR(ram,preg)) /\
    (VAL3(WORD3(V(SEG(5,7)(FETCH_ABBR(ram,preg))))) = 5) /\
    ~bflag /\
    (VAL2(WORD2(V(SEG(8,9)(FETCH_ABBR(ram,preg))))) = 0) /\
    ((VAL4(WORD4(V(SEG(0,3)(FETCH_ABBR(ram,preg))))) = 3) \/
     (VAL4(WORD4(V(SEG(0,3)(FETCH_ABBR(ram,preg))))) = 5) \/
     (VAL4(WORD4(V(SEG(0,3)(FETCH_ABBR(ram,preg))))) = 7)) \/

    ~EL 4(FETCH_ABBR(ram,preg)) /\
    (VAL3(WORD3(V(SEG(5,7)(FETCH_ABBR(ram,preg))))) = 4) /\
    bflag /\
    (VAL2(WORD2(V(SEG(8,9)(FETCH_ABBR(ram,preg))))) = 0) /\
    ((VAL4(WORD4(V(SEG(0,3)(FETCH_ABBR(ram,preg))))) = 3) \/
     (VAL4(WORD4(V(SEG(0,3)(FETCH_ABBR(ram,preg))))) = 5) \/
     (VAL4(WORD4(V(SEG(0,3)(FETCH_ABBR(ram,preg))))) = 7)) \/

    ~EL 4(FETCH_ABBR(ram,preg)) /\
    (VAL3(WORD3(V(SEG(5,7)(FETCH_ABBR(ram,preg))))) = 3) /\
    (VAL2(WORD2(V(SEG(8,9)(FETCH_ABBR(ram,preg))))) = 0) /\
    ((VAL4(WORD4(V(SEG(0,3)(FETCH_ABBR(ram,preg))))) = 3) \/
     (VAL4(WORD4(V(SEG(0,3)(FETCH_ABBR(ram,preg))))) = 5) \/
     (VAL4(WORD4(V(SEG(0,3)(FETCH_ABBR(ram,preg))))) = 7) \/
```

[29]The predicate could be just defined over the memory and program counter, but the other arguments are for uniformity with other predicates.

```
~EL 4(FETCH_ABBR(ram,preg)) /\
(WORD4(V(SEG(0,3)(FETCH_ABBR(ram,preg)))) = #1100) /\
~((VAL3(WORD3(V(SEG(5,7)(FETCH_ABBR(ram,preg))))) = 3) \/
  (VAL3(WORD3(V(SEG(5,7)(FETCH_ABBR(ram,preg))))) = 4) \/
  (VAL3(WORD3(V(SEG(5,7)(FETCH_ABBR(ram,preg))))) = 5)) /\
~(VAL3(WORD3(V(SEG(5,7)(FETCH_ABBR(ram,preg))))) = 6) /\
~(VAL3(WORD3(V(SEG(5,7)(FETCH_ABBR(ram,preg))))) = 7) \/

~EL 4(FETCH_ABBR(ram,preg)) /\
~((VAL3(WORD3(V(SEG(5,7)(FETCH_ABBR(ram,preg))))) = 3) \/
  (VAL3(WORD3(V(SEG(5,7)(FETCH_ABBR(ram,preg))))) = 4) \/
  (VAL3(WORD3(V(SEG(5,7)(FETCH_ABBR(ram,preg))))) = 5)) /\
~(VAL3(WORD3(V(SEG(5,7)(FETCH_ABBR(ram,preg))))) = 7) /\
~(VAL3(WORD3(V(SEG(5,7)(FETCH_ABBR(ram,preg))))) = 6) /\
(VAL2(WORD2(V(SEG(8,9)(FETCH_ABBR(ram,preg))))) = 0) /\
~((VAL4(WORD4(V(SEG(0,3)(FETCH_ABBR(ram,preg))))) = 1) \/
  (VAL4(WORD4(V(SEG(0,3)(FETCH_ABBR(ram,preg))))) = 13) \/
  (VAL4(WORD4(V(SEG(0,3)(FETCH_ABBR(ram,preg))))) = 14) \/
  (VAL4(WORD4(V(SEG(0,3)(FETCH_ABBR(ram,preg))))) = 15)) /\
~(WORD4(V(SEG(0,3)(FETCH_ABBR(ram,preg)))) = #1100)))
```

That is, if the incremented program counter has not spread into its top twelve bits (since it must be usable as a 20-bit address), then there are six choices. (See [8] and [22] for precise details of the codings.) The instruction can indicate a comparison using a literal source; or it can indicate a non-comparison. If the latter, there are five choices:

1. The instruction can indicate the program counter as the destination *if* the boolean flag is false, *with* a false boolean flag, and with a literal source – and a no-op if the boolean flag is true. In that case, the operation indicated can only be a memory-read, an addition with overflow detection, or a subtraction with overflow detection – and no other ALU operation. (The function field values 3, 5 and 7 indicate the operations listed, respectively.)

2. It can be the same as above but with the program counter as the destination *if* the boolean flag is true, with a true boolean flag – and a no-op if it is false.

3. It can be as above but with the program counter as the unconditional destination, and no restiction on the boolean flag.

4. It can indicate the a-, x- or y-register as the destination for a shift operation. (The function field value 12 indicates a shift.)

5. It can indicate the a-, x- or y-register as destination, a literal source, and *any* of the arithmetic-logic operations *except* procedure calls (function field value 1), shifts (function field value 12) or the dis-allowed (spare) operations (function field values 13, 14 and 15).

Similarly, we define condition c_2 to hold if the INDEX operation follows FETCH, c_3 to hold if PRECALL follows FETCH, c_5 if READIO is next, c_6 if READMEM is

next, c7 if STOP is next, c8 if WRITEMEM is next, c9 if WRITEIO is next, and c10 if FETCH follows immediately after FETCH. All of the condition definitions follow immediately from analysis of the major state expression produced by a FETCH operation.

It is first of all necessary to confirm that the nine conditions cover all logical possibilities; otherwise the conditional expression for choosing would be defective, and we would be able to describe an instruction type which the block machine could not handle. Since the conditional followed from the block representation, this would indicate that *it* was in some way flawed – either the representation itself or the basic definitions and Figure 1. We therefore prove:

```
|- c2(ram,preg,areg,xreg,yreg,bflag) \/
   c3(ram,preg,areg,xreg,yreg,bflag) \/
   c4(ram,preg,areg,xreg,yreg,bflag) \/
   c5(ram,preg,areg,xreg,yreg,bflag) \/
   c6(ram,preg,areg,xreg,yreg,bflag) \/
   c7(ram,preg,areg,xreg,yreg,bflag) \/
   c8(ram,preg,areg,xreg,yreg,bflag) \/
   c9(ram,preg,areg,xreg,yreg,bflag) \/
   c10(ram,preg,areg,xreg,yreg,bflag)
```

The proof of this fact is very long and messy because there are so many cases to consider. Because the condition definitions themselves are so long, this fact is most easily proved in an abstracted form (with various assumptions on the abstract variables), then instantiated, and the assumptions proved and dismissed. This is the first example so far of *goal-oriented* proof; we know to start with what we want to prove, but not necessarily the method of proof. The theorem is achieved by applying to the goal (the candidate theorem above) a strategy based on case-analysis. By choosing cases in a clever order, we can minimize the number of cases considered, though it is still very large.

In any case, we now have a predicate c4 which holds exactly when major state 4 follows major state 1 (in any of several ways). The example case of the previous section, however, is more specific: it is the major state 4 in which the comparison field holds the value F and the function selector field holds the word #0101, indicating addition with overflow detection. That is so far consistent with *four* of the ways in which c4 can hold. To simplify the presentation, we further assume that the program counter is *not* the destination (i.e. the destination field is not 3, 4 or 5); otherwise we would have to check whether the computed result exceeded twenty significant bits. With that assumption, there is only one remaining way in which c4 can hold. Therefore we define a more specific predicate to describe the transition to our *particular* major state 4 – i.e. we define a *sub-condition* of c4 called c4_F_5 (to reflect the comparison and function selector fields. We would write c4_F_345_5

for the sub-condition in which the destination*were* 3, 4 or 5). The sub-condition is:

```
|- c4_F_5(ram,preg,areg,xreg,yreg,bflag) =
   (V
    (SEG
     (20,31)
     (BITS32
      (WORD32
       (V(SEG(0,31)(BITS33(WORD33((VAL32(WORD32(VAL20 preg))) + 1)))))))) =
    0) /\
   ~EL 4(FETCH_ABBR(ram,preg)) /\
   ~((VAL3(WORD3(V(SEG(5,7)(FETCH_ABBR(ram,preg))))) = 3) \/
     (VAL3(WORD3(V(SEG(5,7)(FETCH_ABBR(ram,preg))))) = 4) \/
     (VAL3(WORD3(V(SEG(5,7)(FETCH_ABBR(ram,preg))))) = 5)) /\
   ~(VAL3(WORD3(V(SEG(5,7)(FETCH_ABBR(ram,preg))))) = 7) /\
   ~(VAL3(WORD3(V(SEG(5,7)(FETCH_ABBR(ram,preg))))) = 6) /\
   (VAL2(WORD2(V(SEG(8,9)(FETCH_ABBR(ram,preg))))) = 0) /\
   (WORD4(V(SEG(0,3)(FETCH_ABBR(ram,preg)))) = #0101)
```

There are in all thirty-four sub-conditions of `c4`, where for each sub-condition the function selector and comparison indicator are given definite values, and the question of whether the program counter is the destination is definitely resolved. All of the sub-conditions are named in the same way that `c4_F_5` was named; in the shift case (where the function selector is 12), analysis of the conditional expression reveals that the 2-bit memory selection field must be specified also, because it doubles there as an inidicator of differentiate kinds of shifts; hence the name '`c4_F_12_0`', and so on. We prove (much as before) that these thirty-four sub-conditions cover all logical possibilities *within* `c4`:

```
|- c4(ram,preg,areg,xreg,yreg,bflag) =

   c4_T_0(ram,preg,areg,xreg,yreg,bflag) \/
   c4_T_1(ram,preg,areg,xreg,yreg,bflag) \/
   c4_T_2(ram,preg,areg,xreg,yreg,bflag) \/
   c4_T_3(ram,preg,areg,xreg,yreg,bflag) \/
   c4_T_4(ram,preg,areg,xreg,yreg,bflag) \/
   c4_T_5(ram,preg,areg,xreg,yreg,bflag) \/
   c4_T_6(ram,preg,areg,xreg,yreg,bflag) \/
   c4_T_7(ram,preg,areg,xreg,yreg,bflag) \/
   c4_T_8(ram,preg,areg,xreg,yreg,bflag) \/
   c4_T_9(ram,preg,areg,xreg,yreg,bflag) \/
   c4_T_10(ram,preg,areg,xreg,yreg,bflag) \/
   c4_T_11(ram,preg,areg,xreg,yreg,bflag) \/
   c4_T_12(ram,preg,areg,xreg,yreg,bflag) \/
   c4_T_13(ram,preg,areg,xreg,yreg,bflag) \/
   c4_T_14(ram,preg,areg,xreg,yreg,bflag) \/
   c4_T_15(ram,preg,areg,xreg,yreg,bflag) \/

   c4_F_12_0(ram,preg,areg,xreg,yreg,bflag) \/
   c4_F_12_1(ram,preg,areg,xreg,yreg,bflag) \/
   c4_F_12_2(ram,preg,areg,xreg,yreg,bflag) \/
   c4_F_12_3(ram,preg,areg,xreg,yreg,bflag) \/

   c4_F_345_3(ram,preg,areg,xreg,yreg,bflag) \/
   c4_F_345_5(ram,preg,areg,xreg,yreg,bflag) \/
   c4_F_345_7(ram,preg,areg,xreg,yreg,bflag) \/

   c4_F_0(ram,preg,areg,xreg,yreg,bflag) \/
   c4_F_2(ram,preg,areg,xreg,yreg,bflag) \/
   c4_F_3(ram,preg,areg,xreg,yreg,bflag) \/
   c4_F_4(ram,preg,areg,xreg,yreg,bflag) \/
   c4_F_5(ram,preg,areg,xreg,yreg,bflag) \/
   c4_F_6(ram,preg,areg,xreg,yreg,bflag) \/
   c4_F_7(ram,preg,areg,xreg,yreg,bflag) \/
   c4_F_8(ram,preg,areg,xreg,yreg,bflag) \/
   c4_F_9(ram,preg,areg,xreg,yreg,bflag) \/
   c4_F_10(ram,preg,areg,xreg,yreg,bflag) \/
   c4_F_11(ram,preg,areg,xreg,yreg,bflag)
```

The proof, as before, considers a very large number of cases, represented initially in an abstracted way. The first sixteen sub-conditions shown are comparison operations; the next four are shifts; the next three are non-comparisons with the program counter as destination; and the final eleven are non-comparisons with one of the other registers as destination. (Note that among the latter eleven, no sub-condition has a function field value of 1; 1 is reserved for procedure calls, which are considered as a separate major operation, not a variety of PERFORM ALU operation; and 13, 14 and 15 are dis-allowed as values.)

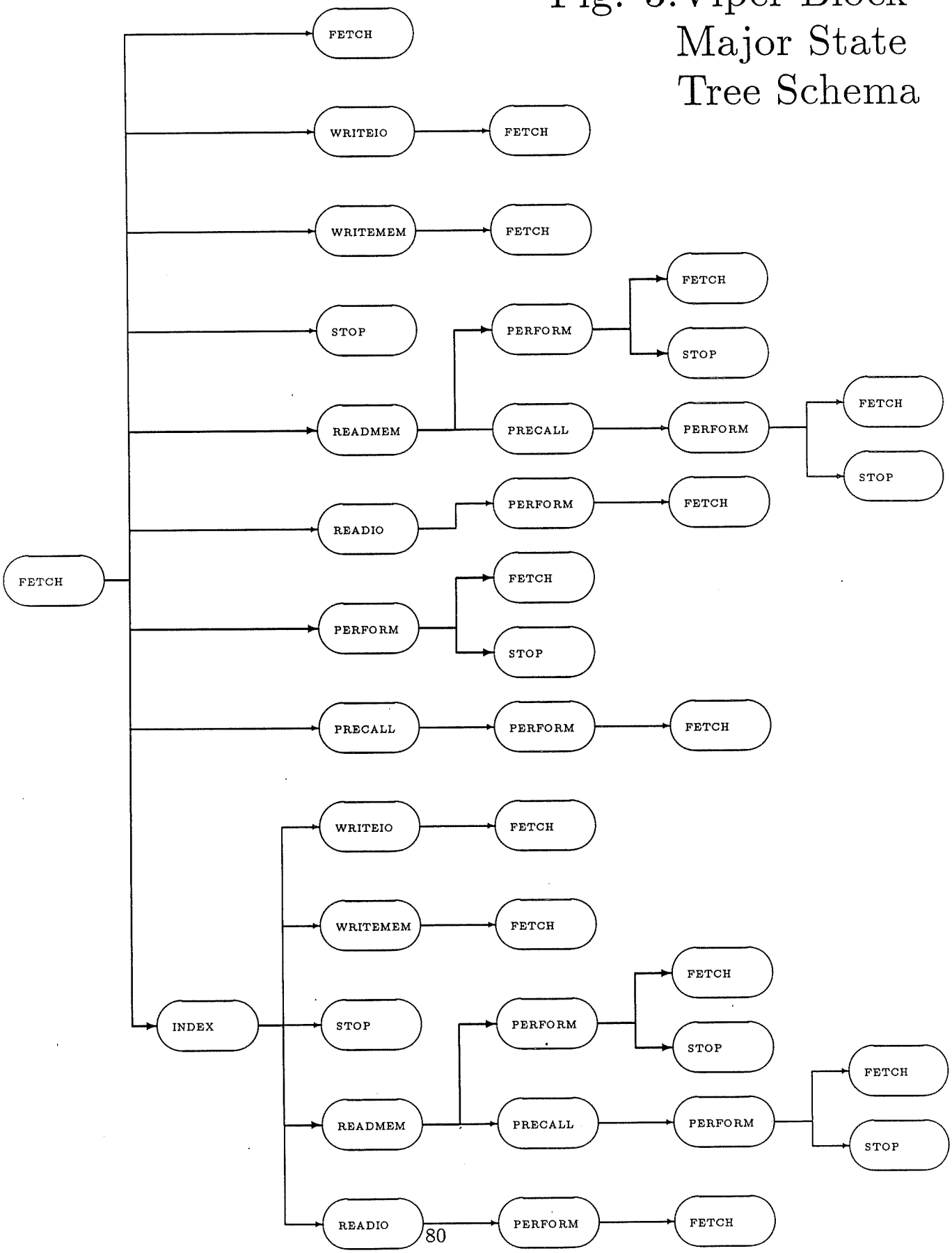
These conditions define thirty-four major states of the PERFORM ALU type, where initially (and in [5]) we had considered just one. Similarly, the PERFORM ALU states that follows a READMEM state or an INDEX *and* a READMEM state (in sequence) also branch into multiple states (twenty-nine each); these are elaborations of the corresponding PERFORM ALU operations in which the source address is either computed or looked up in memory and then computed, rather than just being taken literally.

The complete set of minor state transition theorems tells us that besides the FETCH state, other major states also offer a choice of next state. For example, the INDEX state can be followed by any of the four read or write operations (or by STOP). Also, as in the add-overflow example, the PERFORM ALU state sometimes specifies a choice between STOP or FETCH as the next state. (Had we considered the case in which the function selector were 4, indicating addition *without* overflow checks, there would have been no choice of next states.) In general, STOP-FETCH choices are governed by conditions which we call c17. In the add-overflow case, we define c17_F_5, based on the complicated stopping expression produced in that case, in the context of having previously fetched an instruction. (See Section 9.) The condition holds if the machine does *not* stop. The corresponding condition is just the negation, so proving that all possibilities are covered is easy.

8.2 The Major State Tree

In the analysis of the minor states, it was never the case that at a given minor state there was a choice of next minor state; the sequences were linear. This is not true of major states; the analysis of the way major states follow each other leads to a representation of the major state transitions of the block machine as a tree (or more accurately, a graph). Each possible sequence of major states (i.e. each path through the graph) is represented in Figure 3, below.

Fig. 3:Viper Block Major State Tree Schema



The bolder lines in the figure are path *schemata*, e.g. the collective thirty-or-so paths which follow the pattern of major states shown and contain the PERFORM ALU state. Of these, five or so in each path schema admit the possibility of stopping – e.g. the add-overflow case. The stopping cases are also represented collectively by a bolder line.

For each separate path (not path schema), of which there are one hundred and twenty-two, we can now consider composing the relevant minor state transition theorems to infer cumulative *major* path results. This is done by exactly the same method which yielded the cumulative minor state results.

8.3 Major State Transitions

The example case is continued; all one hundred twenty-two major state sequences are treated similarly.

The final cumulative *minor* state theorems for two nodes are considered: FETCH and ADD OVERFLOW. A recursive procedure, just as before, is used to compose then. The first major state in any sequence is always FETCH, so the procedure uses the minor state cumulative theorem for FETCH as its initial theorem[30].

It is assumed that condition `c4_F_5` holds. We then procede as before, using the case assumption where possible, and stopping the first time the major state becomes 1 again, indicating a fresh FETCH cycle. The possibility of the state becoming 8 (STOP) must also be dealt with; the minor state cumulative theorem for the STOP major state (not shown) asserts that at the final time (time `n+1`)

```
major_sig(n+1) = (reset_sig n => #0010 | #1000)
```

so that the next state may be 2 (RESET) or 8 (STOP), depending on the `reset` signal. The minor state cumulative theorem for the RESET state asserts that at the final time (time `n+4`)

```
major_sig(n+4) = #0001
```

This means that the block machine will remain in the stop state indefinitely, until and unless the `reset` line (connected to the external line `e_resetbar`, as we have seen) becomes set, the next time after which it will move to the RESET state and thence to FETCH again. Since a possibly infinite path cannot conveniently be analyzed, the analyis stops at the point at which the STOP state is reached. In any case,

[30]This is another instance in which the informal description of the Viper block machine is required for the anaylsis; there is nothing in the specifications to suggest that FETCH is special in this way – though of course it would be a sensible guess.

since the high level and major state specifications of Viper do not take resetting into account at all, there would be no point in trying to find a method, using an inductive argument, say, that allowed us to trace all paths back to FETCH.

Suppose that c_17_F_5 holds (the no-stop case). The recursive ML procedure is applied to the initial theorem (the cumulative minor state theorem for FETCH). The procedure then infers the sequence of cumulative theorems for each major state, culminating in the one whose major state points back to FETCH again. In this example, the final theorem is the next one after FETCH: the add-verflow theorem.

To shorten the expressions as results accumulate over major paths, we abbreviated the recurring source and memory sub-expressions:

```
|- REGSELECT_ABBR(areg,xreg,yreg,preg,ram) =
   NOT32
   (REGSELECT
    (areg,xreg,yreg,
     NOT20
     (WORD20
      (V
       (SEG
        (0,19)
        (BITS32
         (NOT32
          (WORD32
           (V
            (SEG(0,31)(BITS33(WORD33((VAL32(WORD32(VAL20 preg))) + 1))))))))))),
    WORD2(V(SEG(10,11)(FETCH_ABBR(ram,preg))))))

  |- MEM_ABBR(ram,preg) =
     WORD32
     (V(SEG(0,19)(BITS32(FETCH21 ram(WORD21(V(CONS F(BITS20 preg))))))))
```

At the end, the a-register's final value is

```
areg_sig(n+12) =
((~EL
   0
   (BITS7
    (DSFPRIM(WORD3(V(SEG(5,7)(FETCH_ABBR(ram_sig n,preg_sig n)))))))) =>
 WORD32
 (V
  (SEG
   (0,31)
   (BITS33
    (WORD33
     ((VAL32
       (REGSELECT_ABBR
        (areg_sig n,xreg_sig n,yreg_sig n,preg_sig n,ram_sig n))) +
      (VAL32(MEM_ABBR(ram_sig n,preg_sig n))))))) |
 areg_sig n)
```

Compared with the result for the addition operation in isolation (page 68), this naturally has the same form; but the value taken from the temporary register (t-register) is the word obtained from memory; the destination field found in the instruction register is the destination field of the looked-up word, as so on – as one would expect.

In the corresponding major sequence *with* overflow (i.e. assuming that c17_F_5 does *not* hold), the final value of the a-register, and so on, is the same as above –

the addition is still performed. However, the stopping value becomes true in this case, and the next major state is 8 (STOP).

As in the minor state transitions, assumptions accumulate when we compose the theorems. Here, assumption sets from the two major states are produced. As a result, we have to have assumed, by the final theorem in the sequence, that `e_errorbar_sig` and `e_resetbar_sig` are true from the initial time through time n+11; that `e_reply_sig` and `e_stepbar_sig` are true from the initial time to time n+5; and the former assumptions about the initial state. That is, some signals have to persist through both major states while others only have to persist only until time n+5, through the fetch operation. The rest only have to be true initially.

8.4 Conclusions about the Major State Transitions

We apply the same method to all one hundred twenty-two major state paths. The procedure is similar for all, though in each case there are small variations which make it difficult at first to design a uniform procedure. The total bulk of the theorems can be imagined; the final result in this relatively simple example is a six-page theorem (pretty-printed), and the complications caused by the occurrence of a READMEM (or an INDEX followed by a READMEM) operation before the addition produce very much longer cumulative results.

Once all the major paths through the graph shown in Figure 3 are derived, the formal analysis is provably complete; there is then a complete, proved picture of the block machine's behaviour on all instruction classes (and an assurance that no instruction classes have been overlooked). This in itself is useful because it could be given to the original designers for inspection; just perusing the results could reveal problems. Furthermore, the proved results could be used as the basis for a simulator.

For the non-ALU sequences, the results are not very complicated and they *appear* to be as intended. Some of the arithmetic-logic paths are also apparently correct. Others, in particular the additions, subtractions and comparisons, are neither obviously correct nor incorrect, and require further study. So far, there do not seem to be any definitely incorrect results, but obviously, since the formal analysis ends at this point, there very well could be. For that reason, a great deal of care should be taken in describing the Viper block model as being 'verified'; it has to date only been *analyzed* as described in this section, and inspected (as described in the next section).

The rest of the analyis is completely speculative: we explore what facts and

equipment would be required to complete the proof up to the point of equivalence with the high level.

9 Speculation on the Rest of the Proof

In Section 6.1, the nature of the state transformations at the specification level was illustrated. We have so far deduced the behaviour of the block machine on every instruction schema, and proved that none was omitted. To complete the proof of correctness of the Viper block model with respect to the high level specification, it would be necessary to compare the high level and block level results for every traversal of the major state tree of the block machine – that is, for over one hundred cases – where each such traversal corresponds to the processing of a single instruction schema. To be able to compare the two sets of results, the logical conditions determining a particular traversal would first have to be related to the conditional choices of the high level specification; then the high level definition would have to be fully unfolded in every case; and finally, the results at the two levels would have to be proved equivalent in some appropriate sense. None of these things has been done, for several reasons:

1. The amount of work involved in proving this number of cases would be considerable, even though there is a certain amount of common effort.

2. Both (i) relating the block's major state path predicates to the high level conditional choices and (ii) relating the block results to the high level results involve a great deal of intricate reasoning about complex bit manipulations. At the time this research was done, there was not enough infrastructure in HOL to support sophisticated reasoning about bits and words; the only method would have been to carry along a large (and possibly inconsistent) set of assumptions about bit-strings. T. Melham [20] at Cambridge is currently developing a definitional framework for bit-string reasoning in HOL. However, it would still remain to build within that theory a library of theorems adequate to support the complex bit-manipulations involved in the equivalence proof.

3. The results at the two levels are quite disparate in the more difficult cases (particularly in cases of ALU operations which are not shifts). Up to the present point in the analysis, the definitions were capable of being unfolded without deep understanding of Viper's design or methods of computation.

However, relating the block results to the high level results could not be approached in the same naive way; it would require extensive co-operation between the verifier and the Viper design team.

Nevertheless, it is possible, and quite useful, to speculate on what would be involved in completing the equivalence proofs.

The general form of the ultimate correctness statement was given in Section 5. Further details of the form can be found in [5]; this applied to a major state machine defined directly, but the form of the statement for the block machine would be very similar. The only forseeable complication is that in the block proof, there are the sets of accumulated assumptions attached to each path result, different in each case, which would probably complicate the task of tying the cases together.

The equivalence proofs are necessarily goal-oriented, unlike most of the proofs described above. That is, it is known what the results are for corresponding cases at the two levels, so one begins with the goal and tries to infer the equavilence by applying proof strategies. Goal-oriented proofs are generally more difficult than the rather deterministic unfolding-style foward proofs seen thus far; they require more insight into the problem on the part of the user.

As mentioned, a large body of facts about bit-strings are required to relate block level and high level terms. These are often phrased rather differently – for example, high level expressions are in terms of 32-bit fetched instructions, while block level expressions are in terms of segments of 12-bit instruction codes placed in various internal registers. Because the facts required are not very profound, we assume that they will eventually form part of a theory of bit-strings, and ignor them by simply equating the register and memory source expressions at the two levels. As in Section 5, the two expressions are be abbreviated respectively as `REGSELECT_ABBR(areg,xreg,yreg,preg,ram)` and `MEM_ABBR(ram,preg)`.

Unfolding the high level definition would be straightforward; it is unfolded in a forward way exactly as the block definitions were unfolded. The really difficult part of the equivalence is not relating the predicates to the conditional choices, but relating the results at the two levels. For example, in the addition-overflow case (where the destination is not the program counter) we have claimed (Section 6.1) that the sum specified at the high level by the function `ALU'` is

```
WORD32
(V
 (TL
  (TL
   (BITS34
    (WORD34
     ((VAL33(SIGNEXT(REGSELECT_ABBR(areg,xreg,yreg,preg,ram)))) +
      (VAL33(SIGNEXT(MEM_ABBR(ram,preg))))))))))
```

where sign extension (SIGNEXT) was defined as the construction of a 33-bit word from a 32-bit word by copying the top bit:

```
|- !w. SIGNEXT w = WORD33(V(CONS(EL 31(BITS32 w))(BITS32 w)))
```

The values of the two sign-extended words are added to form a 34-bit word from which the top two bits are then dropped. On the other hand, the sum we have laboriously inferred at the block level is

```
WORD32
(V
 (SEG
  (0,31)
  (BITS33
   (WORD33
    ((VAL32(REGSELECT_ABBR(areg,xreg,yreg,preg,ram))) +
     (VAL32(MEM_ABBR(ram,preg))))))))
```

which involves adding the values of the two registers to form a 33-bit word and dropping the top (thirty-second) bit. One can convince oneself that the two sums are equivalent, but again, the infrastructure to do this formally in HOL is at present rather limited. Further, the difficulty in this case, and more so in some others, stems from the different methods of computation used at the two levels; here, one level is in terms of a 34-bit sign-extension addition algorithm, while the other is an abstract addition with a carry into the thirty-third bit which is then ignored. The difference is clearer in the case of the overflow expressions; the new overflow condition (the high level stopping flag) at the high level was claimed to be

```
~(EL
  32
  (BITS34
   (WORD34
    ((VAL33(SIGNEXT(REGSELECT_ABBR(areg,xreg,yreg,preg,ram)))) +
     (VAL33(SIGNEXT(MEM_ABBR(ram,preg)))))))) =
  EL
  31
  (BITS34
   (WORD34
    ((VAL33(SIGNEXT(REGSELECT_ABBR(areg,xreg,yreg,preg,ram)))) +
     (VAL33(SIGNEXT(MEM_ABBR(ram,preg))))))))
```

That is, overflow depended on the equality of the thirty-second and thirty-first bits of the 34-bit sum (comprising bits 0 to 33) of the values of the 33-bit sign-extended registers. At the block level the corresponding overflow expression is inferred to be

```
(EL 31(BITS32(REGSELECT_ABBR(areg,xreg,yreg,preg,ram))) =
 EL 31(BITS32(MEM_ABBR(ram,preg)))) /\
~(EL 31(BITS32(REGSELECT_ABBR(areg,xreg,yreg,preg,ram))) =
  EL
  31
  (BITS32
   (WORD32
    (V
     (SEG
      (0,31)
      (BITS33
       (WORD33
        ((VAL32(REGSELECT_ABBR(areg,xreg,yreg,preg,ram))) +
         (VAL32(MEM_ABBR(ram,preg))))))))))
```

Here, the overflow depends on the top bits of the two original registers as well as on the penultimate bit of the 33-bit sum of the values in the two original registers. It is difficult to relate these two expressions even informally, aside from problems of reasoning about bit-strings.

In a typical comparison case, we have claimed that the equality of two registers, at the high level, is tested simply by

```
REGSELECT_ABBR(areg,xreg,yreg,preg,ram) = MEM_ABBR(ram,preg)
```

At the low level, it happens that the expression inferred is

```
(V
 (SEG
  (0,15)
  (BITS32
   (WORD32
    (V
     (SEG
      (0,31)
      (BITS33
       (WORD33
        ((VAL32(REGSELECT_ABBR(areg,xreg,yreg,preg,ram))) +
         ((VAL32(NOT32(MEM_ABBR(ram,preg)))) + 1)))))))) =
 0) /\
(V
 (SEG
  (16,19)
  (BITS32
   (WORD32
    (V
     (SEG
      (0,31)
      (BITS33
       (WORD33
        ((VAL32(REGSELECT_ABBR(areg,xreg,yreg,preg,ram))) +
         ((VAL32(NOT32(MEM_ABBR(ram,preg)))) + 1)))))))) =
 0) /\
(V
 (SEG
  (20,31)
  (BITS32
   (WORD32
    (V
     (SEG
      (0,31)
      (BITS33
       (WORD33
        ((VAL32(REGSELECT_ABBR(areg,xreg,yreg,preg,ram))) +
         ((VAL32(NOT32(MEM_ABBR(ram,preg)))) + 1)))))))) =
 0)
```

Here, the register and memory source are combined by negating the memory source and adding one to its value, then adding that value to the value of the register

source. The thirty-third bit of that sum is dropped, and the 32-bit result is partitioned into three sections, each of which is tested for equality to 0. Again, it takes some thought to convince oneself that the two levels are equiavlent; and only the designers of Viper could explain the computational significance of the block level partition.

For the less-than comparison (source value is less than memory value) the top level result was claimed to be

```
EL
32
(BITS34
 (WORD34
  ((VAL33(SIGNEXT(REGSELECT_ABBR(areg,xreg,yreg,preg,ram)))) +
   ((VAL33(SIGNEXT(MEM_ABBR(ram,preg))) = 0) => 0 |
    (VAL33(NOT33(SIGNEXT(MEM_ABBR(ram,preg))))) + 1))))
```

while at the block level what we derive is

```
~(EL 31(BITS32(REGSELECT_ABBR(areg,xreg,yreg,preg,ram))) =
  EL
  31
  (BITS32
   (WORD32
    (V
     (SEG
      (0,31)
      (BITS33
       (WORD33
        ((VAL32(REGSELECT_ABBR(areg,xreg,yreg,preg,ram))) +
         ((VAL32(NOT32(MEM_ABBR(ram,preg)))) + 1)))))))) /\
~EL 31(BITS32(MEM_ABBR(ram,preg))) \/
EL 31(BITS32(REGSELECT_ABBR(areg,xreg,yreg,preg,ram))) /\
EL
31
(BITS32
 (WORD32
  (V
   (SEG
    (0,31)
    (BITS33
     (WORD33
      ((VAL32(REGSELECT_ABBR(areg,xreg,yreg,preg,ram))) +
       ((VAL32(NOT32(MEM_ABBR(ram,preg)))) + 1)))))))
```

Here, the top bits of the original registers and the top bits of the same sum as before are the determining factors. This is a case in which it is not all easy to see the connection between levels, and not easy to explain the differing methods of computation.

Finally, the procedure call case was said to give the following jump address at the high level:

```
WORD20(V(SEG(0,19)(BITS32(MEM_ABBR(ram,preg)))))
```

At the block level the address is inferred to be:

```
NOT20(WORD20(V(SEG(0,19)(BITS32(NOT32(MEM_ABBR(ram,preg)))))))
```

which is clearly equivalent (once we have proved suitable lemmas about inversion).

The expressions resulting from indirect addressing, or worse, offset addressing, are particularly long and cumbersome, though the ideas should be the same, for both, as in the corresponding simple cases.

In summary, some of the pairs of high and block level expressions are not related in an obvious way, while others require only simple lemmas to be shown equal. The main difficulty rests in understanding and relating the different methods of computation at the two levels.

A minor oddity is that at the block level, two store operations are performed when one would be expected. The high level memory value resulting from this operation is simply

```
STORE21
(WORD21
 (V
  (CONS
   F
   (BITS20(WORD20(V(SEG(0,19)(BITS32(MEM_ABBR(ram,preg))))))))))
(REGSELECT_ABBR(areg,xreg,yreg,preg,ram))
ram
```

whereas at the low level the new memory value is

```
STORE21
(WORD21
 (V
  (CONS
   F
   (BITS20(WORD20(V(SEG(0,19)(BITS32(MEM_ABBR(ram,preg))))))))))
(REGSELECT_ABBR(areg,xreg,yreg,preg,ram))
(STORE21
 (WORD21
  (V
   (CONS
    F
    (BITS20(WORD20(V(SEG(0,19)(BITS32(MEM_ABBR(ram,preg))))))))))
 (REGSELECT_ABBR(areg,xreg,yreg,preg,ram))
 ram)
```

The double store is clearly equivalent to a single store, and there is apparently a reason for this construction[31].

A slightly more puzzling discrepancy is that the stop flag component of the high level (visible) state is, in exactly one case, not equivalent to the block level stop flag. Because of this, the two flags are not identifiable, although one might well expect them to be. The instruction schema on which they differ is the illegal instruction. The high level function specifies that in any of several illegal instruction types, the new state will have all other registers unchanged but the stop flag set to true (see page 29). At the block level, on the other hand, the new value of the stopping flag is determined exclusively by whether the program counter can be

[31]C. Pygott, private communication.

incremented without exceeding twenty bits – and any other illegal conditions are ignored. Because of this fact, the two stopping flags cannot be identified (even though in all other instances they correspond). To circumvent the problem (which seems unintentional), we have to relate the stopping flag at the high level to the equation '`major = 8`' at the block level – i.e. to whether the major state is STOP. This does give a pair of corresponding expressions, but it is rather unaesthetic.

Since the equivalence proof can be explored but cannot reasonably be completed at the present time, the next logical step was for the designers to carefully examine the block results. With suitable abbreviations these are quite readable, and the designers were able to report that the block machine results bore at least a roughly plausible relation to the specified results. This is of course no substitute for a proof, but is a good sign. Nevertheless, it remains possible that a completed formal proof might reveal subtle errors not obvious from an informal scan of the results by the designers. Thus it is really too soon to claim that the Viper block model has been verified, and, as discussed in Section 2, a great deal of care should be taken in making that sort of claim.

10 Lessons and Conclusions

In this section we amplify some of the points made in the Introduction.

This report has described the partially completed proof of correctness of the block model of the Viper microprocessor. The block model is a register-transfer level model of Viper, comprising functional descriptions of the computational units such as the ALU, together with pictorial descriptions of the flow of data and control signals, over time, between units. Viper's correctness is analyzed relative to a higher level specification given in terms of abstract state transformations. The block model was intended to assist in Viper's circuit design phase. Both Viper and the block model were developed at RSRE, by J. Cullyer, C. Pygott and J. Kershaw. The idea of structuring the proof according to the tree of execution paths is due to J. Cullyer.

The project thus far has achieved the following:

- We have extracted a fully formal representation of the block machine based on the original block definitions and diagrams. This has been done in the HOL theorem proving system, in which functions (in the mathematical sense) are regarded as first-class objects of the logic. The block machine itself is expressed as a function, so that one can *infer* its value when applied

to arguments. The representation determines the set of possible execution sequences of the machine.

- Through a process of inference consisting in recursively instantiating the theorems expressing the results of individual major state traversals, we have deduced the cumulative effects of each of the block model's major state sequences. Each sequence in this sense represents the execution of a single instruction type. Each major state traversal is achieved in turn by a sequence of minor events itself having a cumulative effect. Therefore the analysis is hierarchically structured according to major and minor sequences. The individual major and minor state traversal theorems themselves rely on a very large number of lemmas about the basic blocks comprising the block model. These lemmas are all proved by straightforward unfolding of definitions[32].

  In effect, we have symbolically evaluated the machine through each execution path, but with the additional security of having proved that the results follow from the definitions. As a side-effect, this provides a secure basis for building a simulator.

  Assumptions about certain incoming and initial signals limit the cases considered; this is necessary in part because the high level specification is itself incomplete as regards certain inputs to the block model (e.g. externally forced resets).

- We prove further that all possible instruction types have been considered, making the description complete as well as correct.

The fourth and concluding phase of the project, about which we have only speculated, would involve proving a congruence between the block results and the high level results for corresponding instruction types. This phase is impractical to complete at the moment, for several reasons. One reason is that the number of cases and the size of some of the resulting expressions is dauntingly large (for what would appear to be dwindling research interest). A second reason is the current lack of a well-developed HOL infrastructure for supporting advanced reasoning about bit-string manipulations. (T. Melham at Cambridge ([20]) is working on developing an appropriate HOL theory, but it still remains to build within this theory a library of useful theorems leading up to the sort of theorem needed in

[32]By 'unfolding' we mean the application of lambda-expressions to values, followed by beta-reduction.

the equivalence statement – see Section 9.) Finally, it became clear that relating the two disparate descriptions would involve close interaction between verification team and design team, and could not be done as independently as the proof thus far has been.

The success of the proof thus far must be qualified by three caveats, all of which have been discussed in detail in Section 2. Firstly, as just mentioned, the high level specification itself is incomplete as regards Viper as a whole; it describes only the fetch-decode-execute cycle of the machine. Proving that a design satisfies an incomplete specification still leaves plenty of scope for an unacceptable implementation of the design. Secondly, the problem of establishing that the formal representation of the block is itself accurate has no real solution. For example, in the present proof, an incorrect representation was originally derived by cross-naming two block outputs of the same type. The description was subsequently used to generate plausible block results (in most cases) until the error happened to be noticed. Thirdly, the block level itself is still very abstract. Between the block level and the manufactured chip there remain, for example, the gate level and the electronic level – at which problems seem at least as likely to occur. Furthermore, a physical chip cannot be verified in a mathematical sense. Any model is inaccurate to some extent; a 'verified' safety-critical chip can still go wrong and cause loss of life and environment.

As has been described at length, all of the analytical concepts of the proof stem from the functional representation we derived of the block machine. These include the state of the block machine, the shape of the major state tree, and all of the the path predicates. Almost the whole of the proof (with the exception of the proofs that no instruction types have been omitted) consisted in unfolding definitions on typical values. This is not the only mode in which HOL can be used, but it is the easiest, as it requires the user to follow rather than lead the analysis. Indeed, the proof thus far requires minimal understanding of the operation of Viper and its design rationale; all that is really essential is to understand that there are major and minor cycles, and that the major start with FETCH (1), while the minor start with 0. The fourth phase, however, in which the block results are shown to implement the specified results, is different. There, instead of applying routine transformations to known facts to yield unforeseen results, conjectures (goals) are reduced to subgoals by application of carefully planned strategies. These strategies have to embody some understanding of the reason for the implementation, or the proof drifts into useless subgoals.

A brief account of the size of and time required by the proof may be of interest. (It should be borne in mind here that all HOL proofs, like LCF proofs, are actually *performed* to the smallest logical steps; any lemmas used must already have been proved; and no procedures are used to shortcut the process of formal deduction.) The approximate number of primitive inference steps for the various groups of theorems is given below, with the approximate elapsed cpu time on a 12-megabyte Sun-3, and the relevant section of this report (if the issue was discussed).

- Deriving the representation: 95,000 steps (2 hours) (Section 6)
- Lemmas about particular block values for specific minor and/or major states: 5,130,000 steps (50 hours) (Section 7.6)
- Theorems about the transitions through minor states: 210,000 steps (15 hours) (Section 7)
- Proof that all cases have been covered: 160,000 steps (14 hours) (Section 8.1)
- Lemmas about major path predicates: 530,000 steps (7 hours)
- Theorems about the transitions through major states: 128,000 steps (10 hours) (Section 8)
- Projected theorems expressing equivalence to the high level, provided with appropriate lemmas about bit-strings: 900,000 steps (14 hours) (Section 9)
- Total without equivalence proof: 6,253,000 steps (98 hours)
- Total with equivalence proof: about 7,153,000 steps (about 112 hours)

This effort took one person working full-time eleven months to complete. It can be observed that the greatest bulk of the proof by far is in the lemmas which describe the behaviour of the individual blocks on specific major and minor values. These lemmas are all very simple unfoldings; there are just a large number of them, since the block results all have to be inferred for most possible major and minor values. This is encouraging, since that phase involved relatively little time or effort on the part of the user. The most user time was spent in tracing the one-hundred or so execution paths, but again that was a fairly routine process. The most difficult (creative) part of the proof effort was deriving the logical representation of the block machine.

These observations suggest that for a trained user it is not difficult to verify computer designs; it requires more patience perhaps than cleverness. This holds despite the absence of sophisticated proof-editors, friendly interfaces, helpful error messages, or support for proof structuring (beyond a simple subgoal stack). However, it is true that unfolding one hundred-odd cases was tedious. Although the cases were quite similar to one another (especially the sets of corresponding ALU cases using different means of addressing), each seemed to involve some complication which prevented previous strategies from working without some minor change. Generalizing the strategies would be a useful exercise.

The representation methods used in the block proof were all standard in 'Cambridge style' hardware verification circles; the author cannot claim any original contribution. Likewise, the proof strategies deployed (as mentioned) were simple ones – not particularly interesting or original in themselves. Only the size of the proof and the realistic nature of the problem are unique (and probably also the length of some of the theorems).

In terms of the original plan [9] to prove that the block machine implements major state machine (which was already proved [5] to implement the high level specification), it appeared to be more sensible to make the link directly from the block level to the high level. We list some of the issues related to this decision:

- The use of the intermediate level seemed unnecessary, for a start; since equivalence with the top level was the ultimate goal anyway, the need for some sort of transitivity argument was avoided by a direct connection.

- The block and major state levels, though both culminated in major state transition trees, produced rather differently structured trees with different path predicates; the tree and conditions at the block level stem directly from the block representation function, wheras the major state level was designed directly to implement the top level specification. This meant that the local contexts of the trees were difficult to relate. (At the high level, however, the conditional expression branches directly into a state transformation for each possible operation, so there is no problem about local contexts in a tree.)

- The expressions derived at the major state level were never fully unfolded (in particular, ALU expressions were not unfolded) because the property proved did not involve any computation. That is, in the previous major state tree there was only one FETCH-PERFORM path rather than one for each possible ALU operation, so the major state level missed the essence of

the block level: Viper's computational behaviour. This further complicated the use of the major state machine as an intermediate level, as the previous major state results would have to be much further unfolded to be useful.

- Finally, the block level path predicates only emerge from a consideration of the block level major paths, because of the accumulation of results. That is, block level conditions such as c17_F_5 (see page 79) require the whole FETCH-PERFORM sequences to be considered, so that the use of the intermediate level does not seem to get round that stage of analysis (as hoped).

These problems raise the more general issue of using intermediate levels at all, and if so, how they are designed. In a layered approach, it may be difficult to predict, in advance of analyzing the low level in detail, what are the useful concepts at intermediate stages, and what are useful bridging levels. In this case it *seemed* a good idea to isolate the flow of control correctness at the major state level, and the computational problem at the block level – but in fact the block level involved *both* control and computation, and the two were intertwined, so the separation did not work to assist the proof. This is a subject deserving further thought and research. (J. Joyce at Cambridge is also working on this problem [18].) In any case, though not engaged directly, the major state proof was certainly a useful preliminary exercise for the block proof; it established the methodology on a more manageable scale.

Another conclusion, in considering the block proof as a feasibility study in realistically sized proofs, is the performance of the HOL system. HOL does not have a particularly good implementation; it was laced togther from old LCF code which itself has complex historical origins. It is therefore amazing and very encouraging that the system performed so well on a very large proof – orders of magnitude larger than was originally intended when the first LCF system was implemented in the 1970's. This can only be attributed to the successful design principles devised R. Milner, which shine through the layers of implementations and patches. Indeed the only bottleneck in the performance of the block proof was a shortage of disc-space, not the performance of the system or its capacity for handling (parsing, retreiving, manipulating, etc) massive expressions or for managing huge intermediate proof structures.

The massive expressions that comprise the proof arise mostly through the accumulation of effects through layers of processing, as well as through the absence of logical equipment for simplifying expressions about bits and bit-strings. Apart

from that, it would have been better (in retrospect) to have abbreviated the longer recurring sub-expressions as much as possible, for readability and economy; some abbreviation was used, as noted in the paper, but on the whole it was difficult to predict intelligent abbreviations in advance, and after the fact, a bit tedious to re-do the proof for that reason alone.

Re-doing proofs is another issue worth mentioning; at one stage our representation was actually incorrect (involving crossed lines, as mentioned) and it was of course necessary to re-do the proof up to that point. (In fact, this happened a second time because of a typographical error by the author.) The re-doing was facilitated by having preserved the ML code which generated the proof, and simply re-running it on corrected definitions. In fact, since proofs are *performed* rather than produced as sequences of (millions of) steps, the ML code generating the proof is the only tangible end product of a proof effort, and indeed the only 'abstract' description of the proof that one ever has. Two facilities would be desirable in HOL, though both raise philosophical/research problems. One would be a genuinely abstract description of the structure of a proof – more abstract than the sequence of tactics or inference patterns that generate the proof. The other would be a way of tracing 'material dependence' of theorems, so that when it were necessary to go back and change the problem statement, there were a way of knowing which areas of the proof might be or definitely could not be affected. As it stands, the entire proof has to be redone in such cases, which means replacing documentation, and so on.

Many readers will want a simple answer to the question: is Viper correct at the block level? The major state proof [5] was wholly completed, and errors *were* found in the design of the major state model. It may therefore come as a disappointment that the present proof has not been carried far enough for a definite answer to be given; it amounts to an *analysis* of the block machine under all circumstances covered by the specification, but is not a proof that the block model meets the specification. On the positive side, it is now possible for the designers to (i) make a visual inspection of the complete description of the block model's behaviour, and (ii) use the description to simulate the block model in a secure way. These factors may go some way to answering the question. Aside from some confusion about the stopping behaviour of the machine (Section 9), which in any case can be circumvented as described, there does not so far appear to be any problem – that is the best answer we can give.

11 Acknowledgements

Thanks to Mike Gordon who assisted and advised throughout the project, and who helped with this report; and to to the Hardware Verification Group at Cambridge University Computer Laboratory. The work described here was supported by a grant from RSRE under Research Agreement 2029/205(RSRE).

References

[1] H. G. Barrow, VERIFY: A Program for Proving Correctness of Digital Hardware Designs, Artificial Intelligence Vol. 24, 1984

[2] A. Camilieri, M. Gordon and T. Melham, Hardware Verification using Higher-Order Logic, Proceedings of the IFIP WG 10.2 Working Conference: From H.D.L. Descriptions to Guaranteed Correct Circuit Designs, Grenoble, September 1986, ed. D. Borrione, North-Holland, Amsterdam, 1987

[3] A. Church, A Formulation of the Simple Theory of Types, Journal of Symbolic Logic 5, 1940

[4] A. Cohn and M. Gordon, A Mechanized Proof of Correctness of a Simple Counter, University of Cambridge, Computer Laboratory, Tech. Report No. 94, 1986

[5] A. Cohn, A Proof of Correctness of the Viper Microprocessor: the First Level, VLSI Specification, Verification and Synthesis, eds. G. Birtwistle and P.A. Subrahmanyam, Kluwer, 1987; Also University of Cambridge, Computer Laboratory, Tech. Report No. 104

[6] A. Cohn, to appear, Proceedings of the Banff Hardware Verification Workshop, 12-18 June 1988

[7] W. J. Cullyer and C. H. Pygott, Hardware Proofs using LCF_LSM and ELLA, RSRE Memo. 3832, Sept. 1985

[8] W. J. Cullyer, Viper Microprocessor: Formal Specification, RSRE Report 85013, Oct. 1985

[9] W. J. Cullyer, Viper — Correspondence between the Specification and the "Major State Machine", RSRE report No. 86004, Jan. 1986

[10] W. J. Cullyer, Implementing Safety-Critical Systems: The Viper Microprocessor, VLSI Specification, Verification and Synthesis, eds. G. Birtwistle and P.A. Subrahmanyam, Kluwer, 1987

[11] J. Cullyer et. al., forthcoming book

[12] M. Gordon, R. Milner and C. P. Wadsworth, Edinburgh LCF, Lecture Notes in Computer Science No. 78, Springer-Verlag, 1979

[13] M. Gordon, Proving a Computer Correct, University of Cambridge, Computer Laboratory, Tech. Report No. 42, 1983

[14] M. Gordon, HOL: A Machine Oriented Formulation of Higher-Order Logic, University of Cambridge, Computer Laboratory, Tech. Report No. 68, 1985

[15] M. Gordon, HOL: A Proof Generating System for Higher-Order Logic, University of Cambridge, Computer Laboratory, Tech. Report No. 103, 1987; Revised version in VLSI Specification, Verification and Synthesis, eds. G. Birtwistle and P.A. Subrahmanyam, Kluwer, 1987

[16] W. A. Hunt Jr., FM8501: A Verified Microprocessor, University of Texas, Austin, Tech. Report 47, 1985

[17] J. J. Joyce, Formal Verification and Implementation of a Microprocessor, VLSI Specification, Verification and Synthesis, eds. G. Birtwistle and P.A. Subrahmanyam, Kluwer, 1987

[18] J. J. Joyce, private communication

[19] J. Kershaw, Viper: A Microprocessor for Safety-Critical Applications, RSRE Memo. No. 3754, Dec. 1985

[20] T. Melham, private communication

[21] L. Paulson, Interactive Theorem Proving with Cambridge LCF, Cambridge University Press, 1987

[22] C. H. Pygott, Viper: The Electronic Block Model, RSRE Report. No. 86006, July 1986

[23] Viper Microprocessor: Verifiable Integrated Processor for Enhanced Reliability: Development Tools, Charter Technologies Ltd., Publication No. VDT1, Issue 1, Dec. 1987

12 Appendix: The HOL Viper High Level and Block Level Definitions

The following two sections give the complete, corrected RSRE definitions in HOL format of (i) the high level specification and (ii) the blocks of the block model. In each section, the constants (with types) are shown first, followed by the definitions. The block definitions are to be interpreted in conjunction with Figure 1 (page 32). Note that the constants at the two levels are completely distinct; in particular the constant ALU, representing the arithmetic-logic units, occurs at both levels but does not represent the same function in the two cases. This should be regarded as an unfortunate coincidence of names. The state in the high level definitions uses the variable names

```
(ram:mem21_32,p:word20,a:word32,x:word32,y:word32,b:bool,stop:bool)
```

whereas the visible part of the block level state uses

```
(ram:mem21_32,preg:word20,areg:word32,xreg:word32,yreg:word3,bflag:bool,stop:bool)
```

where p corresponds to preg, and similarly for all the other variables, except that the stop component at the high level does not work out to correspond to the stop component at the block level. (See Section 9.)

The definitions are implicitly universally quantified over all free variables.

12.1 The High Level Specification

12.1.1 The Types

```
VALUE:word32#bool#bool->word32
OFLO:word32#bool#bool->bool
CARRY:word32#bool#bool->bool
SVAL:word32#bool#bool->bool
BVAL:word32#bool#bool->bool
TRIM33TO20:word33->word20
TRIM32TO20:word32->word20
TRIM34TO32:word34->word32
PAD20TO32:word20->word32
SIGNEXT:word32->word33
RIGHT:bool#word32->word32
LEFT:word32#bool->word32
RIGHTARITH:word32->word32
NEG:word33->num
ADD32:word32#word32->word32#bool#bool
SUB32:word32#word32->word32#bool#bool
INCP32:word20->word32
FINDINDEX:word2#word32#word32#word32->word32
DEC:(num # num)#word32->num
R:word32->word2
M:word32->word2
D:word32->word3
C:word32->word1
FF:word32->word4
A:word32->word20
NOOP:word3#word1#bool->bool
COMPARE:word4#word32#word32#bool->bool
REG:word2#word32#word32#word32#word20->word32
```

```
INVALID:word32->bool
INSTFETCH:mem21_32 # word20 -> word32"
ALU:word4#word2#word3#word32#word32#bool->word32#bool#bool
WRITE:word3#word1->bool
SPAREFUNC:word3#word1#word4->bool
ILLEGALCALL:word3#word1#word4->bool
ILLEGALPDEST:word3#word1#word4->bool
ILLEGALWRITE:word3#word1#word2->bool
OFFSET:word2#word20#word32#word32->word32
MEMREAD:mem21_32#word2#word20#word32#word32#bool#bool->word32
MEMWRITE:mem21_32#word32#word2#word20#word32#word32#bool->mem21_32
NILM:word3#word1#word4->bool
OUTPUT:word3#word1->bool
INPUT:word3#word1#word4->bool
NEXT:mem21_32#word20#word32#word32#word32#bool#bool->
     mem21_32#word20#word32#word32#word32#bool#bool
```

12.1.2 The Definitions

```
|- VALUE(result,carry,overflow) = result
|- OFLO(result,carry,overflow) = overflow
|- CARRY(result,carry,overflow) = carry
|- SVAL(result,b,abort) = abort
|- BVAL(result,b,abort) = b
|- TRIM33TO20 w = WORD20(V(SEG(0,19)(BITS33 w)))
|- TRIM32TO20 w = WORD20(V(SEG(0,19)(BITS32 w)))
|- TRIM34TO32 w = WORD32(V(TL(TL(BITS34 w))))
|- PAD20TO32 w = WORD32(VAL20 w)
|- SIGNEXT w = (let bitlist = BITS32 w in WORD33(V(CONS(EL 31 bitlist)bitlist)))
|- RIGHT(b,r) = WORD32(V(CONS b(SEG(1,31)(BITS32 r))))
|- LEFT(r,b) = (let twice = V(TL(BITS32 r)) in
                (b => WORD32((twice + twice) + 1) | WORD32(twice + twice)))
|- RIGHTARITH r = (let sign = EL 31(BITS32 r) in
                   WORD32(V(CONS sign(SEG(1,31)(BITS32 r)))))
|- NEG m = ((VAL33 m = 0) => 0 | (VAL33(NOT33 m)) + 1)
|- ADD32(r,m) = (let sum = WORD34((VAL33(SIGNEXT r)) + (VAL33(SIGNEXT m))) in
                 let opposite = (EL 31(BITS32 r)) XOR (EL 31(BITS32 m)) in
                 TRIM34TO32 sum,(EL 32(BITS34 sum)) XOR opposite,
                 (EL 32(BITS34 sum)) XOR (EL 31(BITS34 sum)))
|- SUB32(r,m) = (let dif = WORD34((VAL33(SIGNEXT r)) + (NEG(SIGNEXT m))) in
                 let opposite = (EL 31(BITS32 r)) XOR (EL 31(BITS32 m)) in
                 TRIM34TO32 dif,(EL 32(BITS34 dif)) XOR opposite,
                 (EL 32(BITS34 dif)) XOR (EL 31(BITS34 dif)))
|- INCP32 p = VALUE(ADD32(PAD20TO32 p,WORD32 1))
|- FINDINDEX(msf,t,x,y) = (let xindex = VAL2 msf = 2 in
                           (xindex => VALUE(ADD32(t,x)) | VALUE(ADD32(t,y))))
|- DEC((low,high),w) = V(SEG(low,high)(BITS32 w))
|- R w = WORD2(DEC((30,31),w))
|- M w = WORD2(DEC((28,29),w))
|- D w = WORD3(DEC((25,27),w))
|- C w = WORD1(DEC((24,24),w))
|- FF w = WORD4(DEC((20,23),w))
|- A w = WORD20(DEC((0,19),w))
|- NOOP(dsf,csf,b) = (let df = VAL3 dsf in
                      let cf = VAL1 csf in
                      (cf = 0) AND (((df = 5) AND b) OR ((df = 4) AND (NOT b))))
|- COMPARE(fsf,r,m,b) =
   (let op = VAL4 fsf in
    let dif = WORD34((VAL33(SIGNEXT r)) + (NEG(SIGNEXT m))) in
    let equal = r = m in
    let less = EL 32(BITS34 dif) in
    let borrow = (EL 32(BITS34 dif)) XNOR ((EL 31(BITS32 r)) XNOR (EL 31(BITS32 m))) in
    ((op = 0) => less |
     ((op = 1) => NOT less |
      ((op = 2) => equal |
       ((op = 3) => NOT equal |
        ((op = 4) => less OR equal |
         ((op = 5) => NOT(less OR equal) |
          ((op = 6) => borrow |
           ((op = 7) => NOT borrow |
            ((op = 8) => less OR b |
             ((op = 9) => (NOT less) OR b |
              ((op = 10) => equal OR b |
               ((op = 11) => (NOT equal) OR b |
                ((op = 12) => (less OR equal) OR b |
                 ((op = 13) => (NOT(less OR equal)) OR b |
                  ((op = 14) => borrow OR b | (NOT borrow) OR b)))))))))))))))
|- REG(rsf,a,x,y,p) = (let r = VAL2 rsf in
                       ((r = 0) => a | ((r = 1) => x | ((r = 2) => y | PAD20TO32 p))))
```

```
|- INVALID value = NOT(value = PAD20TO32(TRIM32TO20 value))
|- INSTFETCH(ram,p) = FETCH21 ram(WORD21(V(CONS F(BITS20 p))))
|- ALU(fsf,msf,dsf,r,m,b) =
   (let ff = VAL4 fsf in
    let mf = VAL2 msf in
    let df = VAL3 dsf in
    let pwrite = (df = 3) OR ((df = 4) OR (df = 5)) in
    ((ff = 0) => (NOT32 m,b,pwrite) |
     ((ff = 1) => (m,b,(NOT pwrite) OR (INVALID m)) |
      ((ff = 2) => (m,b,pwrite) |
       ((ff = 3) => (m,b,pwrite AND (INVALID m)) |
        ((ff = 4) => let sum = ADD32(r,m) in VALUE sum,CARRY sum,pwrite |
         ((ff = 5) => let sum = ADD32(r,m) in
                      VALUE sum,b,(OFLO sum) OR (pwrite AND (INVALID(VALUE sum))) |
          ((ff = 6) => let dif = SUB32(r,m) in VALUE dif,CARRY dif,pwrite |
           ((ff = 7) => let dif = SUB32(r,m) in
                        VALUE dif,b,(OFLO dif) OR (pwrite AND (INVALID(VALUE dif))) |
            ((ff = 8) => ((r OR32 m) AND32 (NOT32(r AND32 m)),b,pwrite) |
             ((ff = 9) => (r AND32 m,b,pwrite) |
              ((ff = 10) => (NOT32(r OR32 m),b,pwrite) |
               ((ff = 11) => (r AND32 (NOT32 m),b,pwrite) |
                ((ff = 12) => ((mf = 0) => (RIGHTARITH r,b,pwrite) |
                               ((mf = 1) => (RIGHT(b,r),EL 0(BITS32 r),pwrite) |
                                ((mf = 2) => let double = ADD32(r,r) in
                                             VALUE double,b,(OFLO double) OR pwrite |
                                             (LEFT(r,b),EL 31(BITS32 r),pwrite)))) |
                 ((ff = 13) => (r,b,T) |
                  ((ff = 14) => (r,b,T) | (r,b,T)))))))))))))))
|- WRITE(dsf,csf) = (let df = VAL3 dsf in
                     let cf = VAL1 csf in (cf = 0) AND ((df = 7) OR (df = 6)))
|- SPAREFUNC(dsf,csf,fsf) =
   (let df = VAL3 dsf in
    let cf = VAL1 csf in
    let ff = VAL4 fsf in
    (cf = 0) AND ((NOT((df = 6) OR (df = 7))) AND ((ff = 13) OR ((ff = 14) OR (ff = 15)))))
|- ILLEGALCALL(dsf,csf,fsf) =
   (let df = VAL3 dsf in
    let cf = VAL1 csf in
    let ff = VAL4 fsf in
    (cf = 0) AND ((ff = 1) AND ((df = 0) OR ((df = 1) OR (df = 2)))))
|- ILLEGALPDEST(dsf,csf,fsf) =
   (let df = VAL3 dsf in
    let cf = VAL1 csf in
    let ff = VAL4 fsf in
    (cf = 0) AND (((df = 3) OR ((df = 4) OR (df = 5))) AND
                  (NOT((ff = 1) OR ((ff = 3) OR ((ff = 5) OR (ff = 7))))))
|- ILLEGALWRITE(dsf,csf,msf) = (let mf = VAL2 msf in (WRITE(dsf,csf)) AND (mf = 0))
|- OFFSET(msf,addr,x,y) =
   (let mf = VAL2 msf in
    let addr32 = PAD20TO32 addr in
    ((mf = 0) => addr32 |
     ((mf = 1) => addr32 |
      ((mf = 2) => VALUE(ADD32(addr32,x)) | VALUE(ADD32(addr32,y))))))
|- MEMREAD(ram,msf,addr,x,y,io,nil) =
   (let m = VAL2 msf in
    (nil => WORD32 0 |
     ((m = 0) => PAD20TO32 addr |
      FETCH21
      ram
      (WORD21(V(CONS io(BITS20(TRIM32TO20(OFFSET(msf,addr,x,y))))))))))
|- MEMWRITE(ram,source,msf,addr,x,y,io) =
   (let m = VAL2 msf in
    ((m = 0) =>  ram |
     STORE21
     (WORD21(V(CONS io(BITS20(TRIM32TO20(OFFSET(msf,addr,x,y)))))))
     source
     ram))
|- NILM(dsf,csf,fsf) =
   (let df = VAL3 dsf in
    let cf = VAL1 csf in
    let ff = VAL4 fsf in (cf = 0) AND ((NOT((df = 7) OR (df = 6))) AND (ff = 12)))
|- OUTPUT(dsf,csf) = (let df = VAL3 dsf in let cf = VAL1 csf in (cf = 0) AND (df = 6))
|- INPUT(dsf,csf,fsf) =
   (let df = VAL3 dsf in
    let cf = VAL1 csf in
    let ff = VAL4 fsf in (cf = 0) AND ((NOT((df = 7) OR (df = 6))) AND (ff = 2)))
```

12.1.3 The High Level State Transition Function

```
|- NEXT(ram,p,a,x,y,b,stop) =
   (let instbits = BITS32(INSTFETCH(ram,p)) in
    let newp = TRIM32TO20(INCP32 p) in
    let rsf = WORD2(V(SEG(30,31)instbits)) in
    let msf = WORD2(V(SEG(28,29)instbits)) in
    let dsf = WORD3(V(SEG(25,27)instbits)) in
    let csf = WORD1(V(SEG(24,24)instbits)) in
    let fsf = WORD4(V(SEG(20,23)instbits)) in
    let addr = WORD20(V(SEG(0,19)instbits)) in
    let df = VAL3 dsf in
    let cf = VAL1 csf in
    let ff = VAL4 fsf in
    let comp = cf = 1 in
    let call = (cf = 0) AND (ff = 1) in
    let output = OUTPUT(dsf,csf) in
    let input = INPUT(dsf,csf,fsf) in
    let io = output OR input in
    let writeop = WRITE(dsf,csf) in
    let skip = NOOP(dsf,csf,b) in
    let noinc = INVALID(INCP32 p) in
    let illegaladdr = (NOT(NILM(dsf,csf,fsf))) AND
                      ((INVALID(OFFSET(msf,addr,x,y))) AND
                       (NOT skip)) in
    let illegalcl = ILLEGALCALL(dsf,csf,fsf) in
    let illegalsp = SPAREFUNC(dsf,csf,fsf) in
    let illegalonp = ILLEGALPDEST(dsf,csf,fsf) in
    let illegalwr = ILLEGALWRITE(dsf,csf,msf) in
    let source = REG(rsf,a,x,y,newp) in
    (stop => (ram,p,a,x,y,b,T) |
     ((noinc OR illegaladdr) OR ((illegalcl OR illegalsp) OR (illegalonp OR illegalwr)) =>
      (ram,newp,a,x,y,b,T) |
      (comp => (ram,newp,a,x,y,COMPARE(fsf,source,MEMREAD(ram,msf,addr,x,y,io,F),b),F) |
       (writeop => (MEMWRITE(ram,source,msf,addr,x,y,io),newp,a,x,y,b,F) |
        (skip => (ram,newp,a,x,y,b,F) |
         let m = MEMREAD(ram,msf,addr,x,y,io,NILM(dsf,csf,fsf)) in
         let aluout = ALU(fsf,msf,dsf,source,m,b) in
         ((df = 0) => (ram,newp,VALUE aluout,x,y,BVAL aluout,SVAL aluout) |
          ((df = 1) => (ram,newp,a,VALUE aluout,y,BVAL aluout,SVAL aluout) |
           ((df = 2) => (ram,newp,a,x,VALUE aluout,BVAL aluout,SVAL aluout) |
            (call => (ram,TRIM32TO20(VALUE aluout),a,x,INCP32 p,BVAL aluout,SVAL aluout) |
             (ram,TRIM32TO20(VALUE aluout),a,x,y,BVAL aluout,SVAL aluout)))))))))))
```

12.2 The Block Definitions

Both the types and the definitions are grouped according to the eleven blocks. The types are as follows:

12.2.1 Minor Block Types

```
INCWORD3:word3->word3
NEXTMINOR:bool#bool#bool#bool#word3->word3
```

12.2.2 Major Block Types

```
ILLEGAL_MAJOR:word4->bool
INTRESET:bool#word4->bool
NEXT_MAJOR:bool#bool#word4#word4->word4
FIND_MAJOR:bool#bool#bool#bool#bool#bool#word4#word4->word4
MAJORLOGIC:bool#bool#bool#bool#bool#bool#word4#word4->word4#bool
```

12.2.3 Timeout Block Types

```
NEXT_COUNT:word6#bool#bool->word6
FIND_TIMEOUT:word6#bool->bool
TIMEOUT:word6#bool#bool#bool->word6#bool
```

12.2.4 Timing Block Types

```
FIND_OPTYPE:word4#bool->num
FIND_STA:num#word3->bool
FIND_HALT:num#word3#bool#bool->bool
FIND_STB:num#word3->bool
FIND_RS:num#word3->bool
FIND_NM:num#word3->bool
TIMING:word4#word3#bool#bool#bool->bool#bool#bool#bool#bool
```

12.2.5 BandStop Block Types

```
STOP_LOGIC:word3#bool#bool#bool#bool#bool#bool#bool#bool->bool
B_COMPARE:word4#bool#bool#bool#bool#bool#bool#bool#bool->bool
B_LOGIC:word3#word4#bool#bool#bool#bool#bool#bool#bool#bool#bool#bool#bool->bool
BANDSTOP:word3#word3#word4#bool#bool#word9#bool#bool->bool#bool
```

12.2.6 Decoder Block Types

```
OSEL:word2->word2
DSFPRIM:word3->word7
FIND_CALL:bool#word4->bool
IO_READ:word4->word4
COND_INDEX:word4#bool#word3#bool->word4
ILLEGAL_OP:word4#word3->bool
COND_JUMP:word4#word2->word4
COND_FETCH:word4#bool#word3#word2#bool#bool->word4
FIND_COND:word4#bool#word3#word2#bool#bool#word4->word4
DECODE_PERFORM:word4#word3#word2#word2#bool->word3#word3#word2#bool#word7#word7
DECODE_MAJOR:word4#bool#word3#word2#word2#bool#word4->word3#word3#word2#bool #word7#word7
GET_BM:word3#word3#word2#bool#word7#word7->word3
GET_SM:word3#word3#word2#bool#word7#word7->word3
GET_RS:word3#word3#word2#bool#word7#word7->word2
GET_CI:word3#word3#word2#bool#word7#word7->bool
GET_AL:word3#word3#word2#bool#word7#word7->word7
GET_CL:word3#word3#word2#bool#word7#word7->word7
INST_DECODER:word12#word4#bool#bool->word3#word3#word2#bool#word7#bool#word7#bool#word4#bool
```

12.2.7 ALU Block Types

```
ADD32:word32#word32#bool->word33
ADD32BIT:word32#word32#bool->word32#bool
FIND_SR:bool#bool#bool->bool
BITOP:word32#word32#bool#bool#bool#word7->word32#bool
GET_RM31:bool#word7->bool
GET_AOUT:word32#bool->word32
GET_COUT:word32#bool->bool
ALU:word32#word32#bool#bool#word7->word32#word9
```

12.2.8 Datareg Block Types

```
CLOCK_REGA:word32#word32#bool#bool->word32
CLOCK_REGX:word32#word32#bool#bool->word32
CLOCK_REGY:word32#word32#bool#bool->word32
CLOCK_REGP:word20#word20#bool#bool->word20
CLOCK_REGT:word20#word32#word32#bool#bool#bool#bool->word32
CLOCK_INST:word32#word12#bool#bool->word12
CLOCK_ADDR:word20#word32#word20#bool#bool->word20
REGISTERS:word32#word32#word7#bool#bool#word32#word32#word32#word20#word32#word20#word12->
          word32#word32#word32#word32#word20#word32#word20#word12
REGSELECT:word32#word32#word32#word20#word2->word32
```

12.2.9 FSelect Block Types

```
FSELECT:word12->word4
```

12.2.10 External Block Types

```
FIND_EDB:word32#word4->word32
FIND_DO:word32#bool->word32
EXTERNAL:word32#word20#bool#bool#word4#word32#bool#bool#bool#bool->
         word32#bool#bool#bool#bool#word32#word20#bool#word4#bool#bool#bool#bool#bool#bool
```

12.2.11 Memory Block Types

```
MEMORY:mem21_32#word32#word20#bool#bool#bool->mem21_32#word32
```

The block definitions are as follows:

12.2.12 Minor Block

```
|- INCWORD3 w = (let valw = VAL3 w in ((valw = 7) => WORD3 0 | WORD3(valw + 1)))
|- NEXTMINOR(nextmnbar,advance,reset,intresetbar,minor) =
   (let clear = reset OR ((NOT intresetbar) OR (advance AND (NOT nextmnbar))) in
    (clear => WORD3 0 | (advance => INCWORD3 minor | minor)))
```

12.2.13 Major Block

```
|- ILLEGAL_MAJOR major =
   (let majorval = VAL4 major in
    (majorval = 5) OR
    ((majorval = 8) OR
     ((majorval = 9) OR
      ((majorval = 12) OR
       ((majorval = 13) OR ((majorval = 14) OR (majorval = 15)))))))
|- INTRESET(timeoutbar,major) = (ILLEGAL_MAJOR major) OR (NOT timeoutbar)
|- NEXT_MAJOR(stop,call,cond,major) =
   (let majorval = VAL4 major in
    ((majorval = 0) => cond |
     ((majorval = 1) => cond |
      ((majorval = 2) => WORD4 1 |
       ((majorval = 3) => WORD4 4 |
        ((majorval = 4) => (stop => WORD4 8 | WORD4 1) |
         ((majorval = 6) => WORD4 4 |
          ((majorval = 7) => (call => WORD4 3 | WORD4 4) |
           ((majorval = 10) => WORD4 1 |
            ((majorval = 11) => WORD4 1 | ARB)))))))))))
|- FIND_MAJOR(reset,intreset,advance,nextmain,stop,call,cond,major) =
   (reset => WORD4 2 |
    (intreset => WORD4 8 |
     (advance AND nextmain => NEXT_MAJOR(stop,call,cond,major) | major)))
|- MAJORLOGIC
   (stop,call,timeoutbar,nextmnbar,advance,reset,cond,major) =
   (let intreset = INTRESET(timeoutbar,major) in
    let nextmajor = FIND_MAJOR(reset,intreset,advance,NOT nextmnbar,stop,call,cond,major) in
    nextmajor,NOT intreset)
```

12.2.14 Timeout Block

```
|- NEXT_COUNT(count,reset,strobe) =
   (let countval = VAL6 count in
    (reset OR (NOT strobe) => WORD6 0 | ((countval = 63) => WORD6 0 | WORD6(countval + 1))))
|- FIND_TIMEOUT(count,error) = (count = #111111) OR error
|- TIMEOUT(count,reset,error,strobe) =
   (let nextcount = NEXT_COUNT(count,reset,strobe) in
    let timeoutbar = NOT(FIND_TIMEOUT(count,error)) in nextcount,timeoutbar)
```

12.2.15 Timing Block

```
|- FIND_OPTYPE(major,carryused) =
   (let majorval = VAL4 major in
    ((majorval = 0) => 0 |
     ((majorval = 1) => 3 |
      ((majorval = 2) => 2 |
```

```
      ((majorval = 3) => 2 |
       ((majorval = 4) => (carryused => 0 | 1) |
        ((majorval = 6) => 4 |
         ((majorval = 7) => 4 |
          ((majorval = 8) => 5 |
           ((majorval = 10) => 4 | ((majorval = 11) => 4 | ARB)))))))))))
|- FIND_STA(optype,minor) =
   ((optype = 0) => F |
    ((optype = 1) => F |
     ((optype = 2) => F |
      ((optype = 3) => (minor = #001) |
       ((optype = 4) => (minor = #001) | ((optype = 5) => F | ARB))))))
|- FIND_HALT(optype,minor,pause,reply) =
   (let wait = NOT reply in
    ((optype = 0) => F |
     ((optype = 1) => F |
      ((optype = 2) => F |
       ((optype = 3) =>
        ((minor = #001) AND pause) OR ((minor = #011) AND wait) |
        ((optype = 4) => (minor = #011) AND wait |
         ((optype = 5) => F | ARB)))))))
|- FIND_STB(optype,minor) =
   ((optype = 0) => F |
    ((optype = 1) => F |
     ((optype = 2) => F |
      ((optype = 3) =>
       (minor = #010) OR ((minor = #011) OR (minor = #100)) |
       ((optype = 4) => (minor = #010) OR (minor = #011) |
        ((optype = 5) => F | ARB))))))
|- FIND_RS(optype,minor) =
   ((optype = 0) => (minor = #100) |
    ((optype = 1) => (minor = #011) |
     ((optype = 2) => (minor = #011) |
      ((optype = 3) => (minor = #100) |
       ((optype = 4) => (minor = #011) | ((optype = 5) => F | ARB))))))
|- FIND_NM(optype,minor) =
   ((optype = 0) => (minor = #101) |
    ((optype = 1) => (minor = #100) |
     ((optype = 2) => (minor = #011) |
      ((optype = 3) => (minor = #101) |
       ((optype = 4) => (minor = #011) | ((optype = 5) => F | ARB))))))
|- TIMING(major,minor,pause,reply,carryused) =
   (let optype = FIND_OPTYPE(major,carryused) in
    let advance = NOT(FIND_HALT(optype,minor,pause,reply)) in
    let nextmnbar = NOT(FIND_NM(optype,minor)) in
    let regstrobe = FIND_RS(optype,minor) in
    let strobe = FIND_STB(optype,minor) in
    let strobeaddr = FIND_STA(optype,minor) in advance,nextmnbar,regstrobe,strobe,strobeaddr)
```

12.2.16 BandStop Block

```
 |- STOP_LOGIC
   (stmplx,regstrobe,clkenbp,nztop12,rm31,r31,r30,aout31,stop) =
   (let stopsel = VAL3 stmplx in
    let illegal_jump = clkenbp AND nztop12 in
    let overflow = (r31 = rm31) AND (NOT(r31 = aout31)) in
    (NOT regstrobe => stop |
     ((stopsel = 0) => F |
      ((stopsel = 1) => illegal_jump |
       ((stopsel = 2) => NOT(r31 = r30) |
        ((stopsel = 3) => illegal_jump OR overflow |
         ((stopsel = 4) => T |
          ((stopsel = 5) => nztop12 |
           ((stopsel = 6) => T | ((stopsel = 7) => T | ARB)))))))))
|- B_COMPARE
   (fsf,nzbot16,zmid4,nztop12,coutbar,rm31,r31,aout31,bflag) =
   (let ff = VAL4 fsf in
    let reqm = (NOT nzbot16) AND (zmid4 AND (NOT nztop12)) in
    let rltm = ((NOT(r31 = aout31)) AND rm31) OR (r31 AND aout31) in
    let rgtm = NOT(rltm OR reqm) in
    ((ff = 0) => rltm |
     ((ff = 1) => NOT rltm |
      ((ff = 2) => reqm |
       ((ff = 3) => NOT reqm |
        ((ff = 4) => NOT rgtm |
         ((ff = 5) => rgtm |
          ((ff = 6) => coutbar |
           ((ff = 7) => NOT coutbar |
            ((ff = 8) => bflag OR rltm |
```

```
            ((ff = 9) => bflag OR (NOT rltm) |
             ((ff = 10) => bflag OR reqm |
              ((ff = 11) => bflag OR (NOT reqm) |
               ((ff = 12) => bflag OR (NOT rgtm) |
                ((ff = 13) => bflag OR rgtm |
                 ((ff = 14) => bflag OR coutbar |
                  ((ff = 15) => bflag OR (NOT coutbar) | ARB)))))))))))))))
|- B_LOGIC
   (bmplx,fsf,regstrobe,nzbot16,zmid4,nztop12,coutbar,rm31,r31,r0,aout31,bflag) =
   (let bsel = VAL3 bmplx in
    (NOT regstrobe => bflag |
     ((bsel = 0) => bflag |
      ((bsel = 1) => r0 |
       ((bsel = 2) => r31 |
        ((bsel = 3) => NOT coutbar |
         ((bsel = 4) => coutbar |
          ((bsel = 5) => F |
           ((bsel = 6) =>
            B_COMPARE
            (fsf,nzbot16,zmid4,nztop12,coutbar,rm31,r31,aout31,bflag) |
            ((bsel = 7) => bflag | ARB)))))))))))
|- BANDSTOP
   (stmplx,bmplx,fsf,regstrobe,clkenbp,conditions,stop,bflag) =
   (let nzbot16 = EL 0(BITS9 conditions) in
    let zmid4 = EL 1(BITS9 conditions) in
    let nztop12 = EL 2(BITS9 conditions) in
    let coutbar = EL 3(BITS9 conditions) in
    let rm31 = EL 4(BITS9 conditions) in
    let r31 = EL 5(BITS9 conditions) in
    let r30 = EL 6(BITS9 conditions) in
    let r0 = EL 7(BITS9 conditions) in
    let aout31 = EL 8(BITS9 conditions) in
    let nextstop = STOP_LOGIC(stmplx,regstrobe,clkenbp,nztop12,rm31,r31,r30,aout31,stop) in
    let nextbflag = B_LOGIC
                    (bmplx,fsf,regstrobe,nzbot16,zmid4,nztop12,coutbar,rm31,r31,r0,
                     aout31,bflag) in
    nextstop,nextbflag)
```

12.2.17 Decoder Block

```
|- OSEL msf =
   (let msfval = VAL2 msf in ((msfval = 2) => #01 | ((msfval = 3) => #10 | ARB)))
|- DSFPRIM dsf =
   (let dsfval = VAL3 dsf in
    ((dsfval = 0) => #0111110 |
     ((dsfval = 1) => #0111011 |
      ((dsfval = 2) => #0110111 |
       ((dsfval = 3) => #0111101 |
        ((dsfval = 4) => #0111101 |
         ((dsfval = 5) => #0111101 | ARB)))))))
|- FIND_CALL(csf,fsf) = (NOT csf) AND (fsf = #0001)
|- IO_READ fsf = ((fsf = #0010) => WORD4 6 | WORD4 7)
|- COND_INDEX(fsf,csf,dsf,stop) =
   (let io_read = IO_READ fsf in
    let dsfval = VAL3 dsf in
    (stop => WORD4 8 |
     (csf => WORD4 7 |
      ((dsfval = 7) => WORD4 10 |
       ((dsfval = 6) => WORD4 11 | io_read)))))
|- ILLEGAL_OP(fsf,dsf) =
   (let ff = VAL4 fsf in
    let df = VAL3 dsf in
    let pdest = (df = 3) OR ((df = 4) OR (df = 5)) in
    let legal_on_p = (ff = 1) OR ((ff = 3) OR ((ff = 5) OR (ff = 7))) in
    let illegal_not_p = (ff = 1) OR ((ff = 13) OR ((ff = 14) OR (ff = 15))) in
    ((NOT pdest) AND illegal_not_p) OR (pdest AND (NOT legal_on_p)))
|- COND_JUMP(fsf,msf) =
   (let mf = VAL2 msf in
    let shift = fsf = #1100 in
    let call = fsf = #0001 in
    let io = fsf = #0010 in
    (shift =>  WORD4 4 |
     ((mf = 0) => (call => WORD4 3 | WORD4 4) |
      ((mf = 1) => (io => WORD4 6 | WORD4 7) | WORD4 0))))
|- COND_FETCH(fsf,csf,dsf,msf,bflag,stop) =
   (let cond_jump = COND_JUMP(fsf,msf) in
    let illegal_op = ILLEGAL_OP(fsf,dsf) in
    let mf = VAL2 msf in
    let df = VAL3 dsf in
```

```
    (stop => WORD4 8 |
     (csf =>
      ((mf = 0) => WORD4 4 | ((mf = 1) => WORD4 7 | WORD4 0)) |
      ((df = 7) => ((mf = 0) => WORD4 8 | ((mf = 1) => WORD4 10 | WORD4 0)) |
       ((df = 6) => ((mf = 0) => WORD4 8 | ((mf = 1) => WORD4 11 | WORD4 0)) |
        (illegal_op => WORD4 8 |
         ((df = 5) => (bflag => WORD4 1 | cond_jump) |
          ((df = 4) => (NOT bflag => WORD4 1 | cond_jump) | cond_jump))))))))
|- FIND_COND(fsf,csf,dsf,msf,bflag,stop,major) =
   ((major = #0000) => COND_INDEX(fsf,csf,dsf,stop) |
    ((major = #0001) => COND_FETCH(fsf,csf,dsf,msf,bflag,stop) | ARB))
|- DECODE_PERFORM(fsf,dsf,msf,rsf,bflag) =
   (let bmp_b = #000 in
    let bmp_lsbr = #001 in
    let bmp_msbr = #010 in
    let bmp_carry = #011 in
    let bmp_borrw = #100 in
    let bmp_0 = #101 in
    let bmp_comp = #110 in
    let smp_0 = #000 in
    let smp_nsf = #001 in
    let smp_shfov = #010 in
    let smp_ovf = #011 in
    let smp_1 = #100 in
    let smp_tail = #101 in
    let rsel_x = #00 in
    let cin_t = T in
    let cin_f = F in
    let cin_x = F in
    let alu_and = #1101001 in
    let alu_rmb = #1111001 in
    let alu_zero = #1100000 in
    let alu_m = #1101000 in
    let alu_comm = #1111000 in
    let alu_r = #1110001 in
    let alu_sr = #1110010 in
    let alu_xor = #0111001 in
    let alu_nor = #0111100 in
    let alu_add = #0001001 in
    let alu_sub = #0011001 in
    let alu_incr = #0000001 in
    let alu_sl = #0000101 in
    let fsfval = VAL4 fsf in
    let msfval = VAL2 msf in
    let dsfprim = DSFPRIM dsf in
    ((fsfval = 0) => (bmp_b,smp_0,rsel_x,cin_x,alu_comm,dsfprim) |
     ((fsfval = 1) =>
      (bmp_b,smp_nsf,rsel_x,cin_x,alu_m,dsfprim) |
      ((fsfval = 2) => (bmp_b,smp_0,rsel_x,cin_x,alu_m,dsfprim) |
       ((fsfval = 3) => (bmp_b,smp_nsf,rsel_x,cin_x,alu_m,dsfprim) |
        ((fsfval = 4) => (bmp_carry,smp_0,rsf,cin_f,alu_add,dsfprim) |
         ((fsfval = 5) => (bmp_b,smp_ovf,rsf,cin_f,alu_add,dsfprim) |
          ((fsfval = 6) => (bmp_borrw,smp_0,rsf,cin_t,alu_sub,dsfprim) |
           ((fsfval = 7) => (bmp_b,smp_ovf,rsf,cin_t,alu_sub,dsfprim) |
            ((fsfval = 8) => (bmp_b,smp_0,rsf,cin_x,alu_xor,dsfprim) |
             ((fsfval = 9) => (bmp_b,smp_0,rsf,cin_x,alu_and,dsfprim) |
              ((fsfval = 10) => (bmp_b,smp_0,rsf,cin_x,alu_nor,dsfprim) |
               ((fsfval = 11) => (bmp_b,smp_0,rsf,cin_x,alu_rmb,dsfprim) |
                ((fsfval = 12) =>
                 ((msfval = 0) => (bmp_b,smp_0,rsf,cin_t,alu_sr,dsfprim) |
                  ((msfval = 1) => (bmp_lsbr,smp_0,rsf,cin_f,alu_sr,dsfprim) |
                   ((msfval = 2) => (bmp_b,smp_shfov,rsf,cin_f,alu_sl,dsfprim) |
                    ((msfval = 3) => (bmp_msbr,smp_0,rsf,bflag,alu_sl,dsfprim) |
                     ARB)))) |
                ARB))))))))))))))
|- DECODE_MAJOR(fsf,csf,dsf,msf,rsf,bflag,major) =
   (let bmp_b = #000 in
    let bmp_lsbr = #001 in
    let bmp_msbr = #010 in
    let bmp_carry = #011 in
    let bmp_borrw = #100 in
    let bmp_0 = #101 in
    let bmp_comp = #110 in
    let smp_0 = #000 in
    let smp_nsf = #001 in
    let smp_shfov = #010 in
    let smp_ovf = #011 in
    let smp_1 = #100 in
    let smp_tail = #101 in
    let reg_p = #11 in
    let rsel_x = #00 in
```

```
    let cin_t = T in
    let cin_f = F in
    let cin_x = F in
    let alu_and = #1101001 in
    let alu_rmb = #1111001 in
    let alu_zero = #1100000 in
    let alu_m = #1101000 in
    let alu_comm = #1111000 in
    let alu_r = #1110001 in
    let alu_sr = #1110010 in
    let alu_xor = #0111001 in
    let alu_nor = #0111100 in
    let alu_add = #0001001 in
    let alu_sub = #0011001 in
    let alu_incr = #0000001 in
    let alu_sl = #0000101 in
    let dest_t = #1011111 in
    let dest_y = #0110111 in
    let dest_fet = #0001101 in
    let dest_all = #0110000 in
    let dest_te = #0011111 in
    let dest_non = #0111111 in
    let majorval = VAL4 major in
    let osel = OSEL msf in
    ((majorval = 0) => (bmp_b,smp_tail,osel,cin_f,alu_add,dest_t) |
     ((majorval = 1) => (bmp_b,smp_nsf,reg_p,cin_t,alu_incr,dest_fet) |
      ((majorval = 2) => (bmp_0,smp_0,rsel_x,cin_x,alu_zero,dest_all) |
       ((majorval = 3) => (bmp_b,smp_0,reg_p,cin_x,alu_r,dest_y) |
        ((majorval = 6) => (bmp_b,smp_0,rsel_x,cin_x,alu_zero,dest_te) |
         ((majorval = 7) => (bmp_b,smp_0,rsel_x,cin_x,alu_zero,dest_te) |
          ((majorval = 8) => (bmp_b,smp_0,rsel_x,cin_x,alu_zero,dest_non) |
           ((majorval = 10) => (bmp_b,smp_0,rsf,cin_x,alu_zero,dest_non) |
            ((majorval = 11) => (bmp_b,smp_0,rsf,cin_x,alu_zero,dest_non) |
             ((majorval = 4) => (csf =>
                                 (bmp_comp,smp_0,rsf,cin_t,alu_sub,dest_non) |
                                 DECODE_PERFORM(fsf,dsf,msf,rsf,bflag)) | ARB)))))))))))
|- GET_BM(bmplx,stmplx,rsel,cin,alucon,clkenb) = bmplx
|- GET_SM(bmplx,stmplx,rsel,cin,alucon,clkenb) = stmplx
|- GET_RS(bmplx,stmplx,rsel,cin,alucon,clkenb) = rsel
|- GET_CI(bmplx,stmplx,rsel,cin,alucon,clkenb) = cin
|- GET_AL(bmplx,stmplx,rsel,cin,alucon,clkenb) = alucon
|- GET_CL(bmplx,stmplx,rsel,cin,alucon,clkenb) = clkenb
|- INST_DECODER(inst,major,bflag,stop) =
   (let instlist = BITS12 inst in
    let fsf = WORD4(V(SEG(0,3)instlist)) in
    let csf = EL 4 instlist in
    let dsf = WORD3(V(SEG(5,7)instlist)) in
    let msf = WORD2(V(SEG(8,9)instlist)) in
    let rsf = WORD2(V(SEG(10,11)instlist)) in
    let decode_major = DECODE_MAJOR(fsf,csf,dsf,msf,rsf,bflag,major) in
    let bmplx = GET_BM decode_major in
    let stmplx = GET_SM decode_major in
    let rsel = GET_RS decode_major in
    let cin = GET_CI decode_major in
    let alucon = GET_AL decode_major in
    let clkenb = GET_CL decode_major in
    let carryused = NOT(EL 5(BITS7 alucon)) in
    let clkenbp = NOT(EL 1(BITS7 clkenb)) in
    let cond = FIND_COND(fsf,csf,dsf,msf,bflag,stop,major) in
    let call = FIND_CALL(csf,fsf) in
    bmplx,stmplx,rsel,cin,alucon,carryused,clkenb,clkenbp,cond,call)
```

12.2.18 ALU Block

```
|- ADD32(r,t,cin) = (cin => WORD33((VAL32 r) + ((VAL32 t) + 1)) | WORD33((VAL32 r) + (VAL32 t)))
|- ADD32BIT(r,t,cin) =
   (let sum33 = ADD32(r,t,cin) in
    let aout = WORD32(V(SEG(0,31)(BITS33 sum33))) in
    let carry = EL 32(BITS33 sum33) in aout,carry)
|- FIND_SR(cin,bflag,r31) = (cin => r31 | bflag)
|- BITOP(r,t,bflag,r31,cin,op) =
   (let cout_x = F in
    let tbar = NOT32 t in
    let rbar = NOT32 r in
    let sr = FIND_SR(cin,bflag,r31) in
    let r_xor_t = (r AND32 tbar) OR32 (rbar AND32 t) in
    let shift_right = WORD32(V(CONS sr(SEG(1,31)(BITS32 r)))) in
    ((op = #1101001) => (r AND32 t,cout_x) |
     ((op = #1111001) => (r AND32 tbar,cout_x) |
```

```
      ((op = #1100000) => (WORD32 0,cout_x) |
       ((op = #1101000) => (t,cout_x) |
        ((op = #1111000) => (tbar,cout_x) |
         ((op = #1110001) => (r,cout_x) |
          ((op = #1110010) => (shift_right,cout_x) |
           ((op = #0111001) => (r_xor_t,cout_x) |
            ((op = #0111100) => (NOT32(r OR32 t),cout_x) |
             ((op = #0000101) => ADD32BIT(r,r,cin) |
              ((op = #0001001) => ADD32BIT(r,t,cin) |
               ((op = #0011001) => ADD32BIT(r,tbar,cin) |
                ((op = #0000001) => ADD32BIT(r,WORD32 0,cin) | ARB))))))))))))
|- GET_RM31(t31,alucon) = ((alucon = #0011001) => NOT t31 |
                          ((alucon = #0001001) => t31 | F))
|- GET_AOUT(aout,cout) = aout
|- GET_COUT(aout,cout) = cout
|- ALU(rbar,treg,cin,bflag,alucon) =
   (let r = NOT32 rbar in
    let t31 = EL 31(BITS32 treg) in
    let rm31 = GET_RM31(t31,alucon) in
    let r0 = EL 0(BITS32 r) in
    let r30 = EL 30(BITS32 r) in
    let r31 = EL 31(BITS32 r) in
    let a_c = BITOP(r,treg,bflag,r31,cin,alucon) in
    let aout = GET_AOUT a_c in
    let aoutbar = NOT32 aout in
    let coutbar = NOT(GET_COUT a_c) in
    let aout31 = EL 31(BITS32 aout) in
    let nzbot16 = NOT(V(SEG(0,15)(BITS32 aout)) = 0) in
    let zmid4 = V(SEG(16,19)(BITS32 aout)) = 0 in
    let nztop12 = NOT(V(SEG(20,31)(BITS32 aout)) = 0) in
    let conditions = WORD9(V[aout31;r0;r30;r31;rm31;coutbar;nztop12;zmid4;nzbot16]) in
    aoutbar,conditions)
```

12.2.19 Datareg Block

```
|- CLOCK_REGA(aoutbar,areg,clkenba,regstrobe) = (clkenba AND regstrobe => NOT32 aoutbar | areg)
|- CLOCK_REGX(aoutbar,xreg,clkenbx,regstrobe) = (clkenbx AND regstrobe => NOT32 aoutbar | xreg)
|- CLOCK_REGY(aoutbar,yreg,clkenby,regstrobe) = (clkenby AND regstrobe => NOT32 aoutbar | yreg)
|- CLOCK_REGP(aoutbar_ls20,preg,clkenbp,regstrobe) =
   (clkenbp AND regstrobe => NOT20 aoutbar_ls20 | preg)
|- CLOCK_REGT
   (aoutbar_ls20,extdata,treg,clkenbt,regstrobe,tmplxcon,clkenbinst) =
   (let extdata_ls20 = WORD32(V(SEG(0,19)(BITS32 extdata))) in
    let aout_tail = WORD32(VAL20(NOT20 aoutbar_ls20)) in
    (NOT(clkenbt AND regstrobe) => treg |
     ((NOT tmplxcon) AND (NOT clkenbinst) => extdata |
      (tmplxcon AND (NOT clkenbinst) => aout_tail |
       ((NOT tmplxcon) AND clkenbinst => extdata_ls20 | ARB)))))
|- CLOCK_INST(extdata,inst,clkenbinst,regstrobe) =
   (let extdata_ms12 = WORD12(V(SEG(20,31)(BITS32 extdata))) in
    (clkenbinst AND regstrobe => extdata_ms12 | inst))
|- CLOCK_ADDR(preg,treg,addr,clkenbinst,strobeaddr) =
   (NOT strobeaddr => addr |
    (clkenbinst => preg | WORD20(V(SEG(0,19)(BITS32 treg)))))
|- REGISTERS
   (extdatabar,aoutbar,clkenb,regstrobe,strobeaddr,areg,xreg,yreg,preg,treg,addr,inst) =
   (let extdata = NOT32 extdatabar in
    let aoutbar_ls20 = WORD20(V(SEG(0,19)(BITS32 aoutbar))) in
    let clkenba = NOT(EL 0(BITS7 clkenb)) in
    let clkenbp = NOT(EL 1(BITS7 clkenb)) in
    let clkenbx = NOT(EL 2(BITS7 clkenb)) in
    let clkenby = NOT(EL 3(BITS7 clkenb)) in
    let clkenbinst = NOT(EL 4(BITS7 clkenb)) in
    let clkenbt = NOT(EL 5(BITS7 clkenb)) in
    let tmplxcon = EL 6(BITS7 clkenb) in
    let new_areg = CLOCK_REGA(aoutbar,areg,clkenba,regstrobe) in
    let new_xreg = CLOCK_REGX(aoutbar,xreg,clkenbx,regstrobe) in
    let new_yreg = CLOCK_REGY(aoutbar,yreg,clkenby,regstrobe) in
    let new_preg = CLOCK_REGP(aoutbar_ls20,preg,clkenbp,regstrobe) in
    let new_inst = CLOCK_INST(extdata,inst,clkenbinst,regstrobe) in
    let new_treg = CLOCK_REGT
                   (aoutbar_ls20,extdata,treg,clkenbt,regstrobe,tmplxcon,clkenbinst) in
    let new_addr = CLOCK_ADDR(preg,treg,addr,clkenbinst,strobeaddr) in
    new_areg,new_xreg,new_yreg,new_preg,new_treg,new_addr,new_inst)
|- REGSELECT(areg,xreg,yreg,preg,rsel) =
   (let rf = VAL2 rsel in
    ((rf = 0) => NOT32 areg |
     ((rf = 1) => NOT32 xreg |
      ((rf = 2) => NOT32 yreg |
```

```
    ((rf = 3) => NOT32(WORD32(VAL20 preg)) | ARB)))))
```

12.2.20 FSelect Block

```
|- FSELECT inst = WORD4(V(SEG(0,3)(BITS12 inst)))
```

12.2.21 External Block

```
|- FIND_EDB(e_data_in,major) =
   (let unused_input = WORD32 0 in
    let majorval = VAL4 major in
    ((majorval = 1) => NOT32 e_data_in |
     ((majorval = 6) => NOT32 e_data_in |
      ((majorval = 7) => NOT32 e_data_in | unused_input))))
|- FIND_DO(rbar,writegate) = (let tri_state = WORD32 0 in (writegate => NOT32 rbar | tri_state))
|- EXTERNAL
   (rbar,addr,bflag,strobe,major,e_data_in,e_resetbar,e_errorbar,e_stepbar,e_reply) =
   (let writegate = (major = #1010) OR (major = #1011) in
    FIND_EDB(e_data_in,major),NOT e_resetbar,NOT e_errorbar,NOT e_stepbar,e_reply,
    FIND_DO(rbar,writegate),addr,bflag,major,NOT strobe,(major = #1000),(major = #0100),
    NOT(major = #0001),NOT((major = #0110) OR (major = #1011)),
    NOT((major = #1010) OR (major = #1011)))
```

12.2.22 Memory Block

```
|- MEMORY(ram,e_data_out,e_address,e_iobar,e_writebar,e_strobebar) =
   (let address = WORD21(V(CONS(NOT e_iobar)(BITS20 e_address))) in
    (e_strobebar => (ram,e_data_out) |
     (e_writebar => (ram,FETCH21 ram address) |
      (STORE21 address e_data_out ram,e_data_out))))
```

Index