Formal Verification of a Microcoded VIPER Microprocessor Using HOL

Karl Levitt, Tejkumar Arora, Tony Leung, Sara Kalvala, E. Thomas Schubert, Philip Windley, and Mark Heckman
University of California
Davis, California

Gerald C. Cohen
Boeing Defense & Space Group
Seattle, Washington

Prepared for
Langley Research Center
under Contract NAS1-18586
Preface

This document was generated in support of NASA contract NAS1-18586, Design and Validation of Digital Flight Control Systems Suitable for Fly-By-Wire Applications, Task Assignment 3. Task 3 is associated with formal verification of embedded systems.

The formal verification of a microprocessor involves demonstrating that a specification of the microprocessor is satisfied by its implementation. The specification is usually a formal description of the microprocessor's instructions. Any more concrete description of the microprocessor can suffice for the implementation, but it has become the practice for the implementation to represent the major electronic blocks that constitute the microprocessor (ALU, registers, latches, memory, etc), hence the name electronic block model. Although not necessarily routinely, a realization of the electronic block can be checked by simulation or other testing methods.

A particular microprocessor of interest is Viper, designed by the Royal Signals and Radar Establishment, UK (RSRE) for critical applications. An initial successful proof of Viper (by Avra Cohn) was of its major state model. However, what was verified is considered to be too abstract for an implementation. A subsequent effort was undertaken by Cohn to verify Viper's electronic block model. Both of these efforts made use of the HOL (the Cambridge Higher Order Logic) theorem prover. This latter proof was not completed, mostly because it became too time consuming.

Our view of the incomplete proof of Viper is that the jump in abstraction between the electronic block model and the specification is too great. By introducing intermediate levels between the two extreme models, the overall proof becomes one of establishing more but simpler proofs. Windley, in a recent U. C. Davis PhD thesis showed that the levels can be represented as interpreters, each of which models an abstraction of a microprocessor. For example, one of the levels is an interpreter for the execution of microinstructions. To further simplify the proof effort, Windley developed a theory of generic interpreters--a notation that is sufficiently powerful to represent a large class of interpreters. The interpreter theory has been formalized using generic theories in HOL for use in specifying and verifying microprocessors. The generic interpreter theory formally defines an interpreter and generates a correctness theorem for the generic model stating what it means, in general, for an instance of the interpreter to be correctly implemented.

To demonstrate the effectiveness of this theory on a real microprocessor instruction set, this report presents our results on applying the generic interpreter methodology to Viper. We redesigned Viper as a hierarchy of five interpreters, each of which is an instance of the generic interpreter. The top level specifies the Viper
instruction set, and the lowest is of the abstraction of the conventional electronic block model, but one that implements a microinstruction interpreter.

In this report we discuss our design of the microcoded machine that realizes the Viper instruction set, and our verification of this machine. The design and most of the verification was carried out in 1 person-year by two Master's students with no previous background in formal methods. We also discuss features of the original Viper design that our verification effort does not consider.

The NASA technical monitor for this work is Sally Johnson of the NASA Langley Research Center, Hampton, Virginia.

The work was accomplished at Boeing Military Airplanes, Seattle, Washington, and the University of California, Davis, California. Personnel responsible for the work include:

Boeing Military Airplanes:
D. Gangsaas, Responsible Manager
T. M. Richardson, Program Manager
G. C. Cohen, Principal Investigator

University of California:
Dr. K. Levitt, Chief Researcher
Tejkumar Arora
Tony Leung
Sara Kalvala
E. Thomas Schubert
Dr. Philip Windley
Mark Heckman
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 VIPER</td>
<td>3</td>
</tr>
<tr>
<td>1.2 Abstraction</td>
<td>5</td>
</tr>
<tr>
<td>1.2.1 Hierarchical Decomposition</td>
<td>5</td>
</tr>
<tr>
<td>1.2.2 Generic Interpreters</td>
<td>6</td>
</tr>
<tr>
<td>1.3 What we have accomplished vis-a-vis VIPER</td>
<td>7</td>
</tr>
<tr>
<td>1.4 Notation and Conventions</td>
<td>9</td>
</tr>
<tr>
<td>1.5 Chapter Summaries</td>
<td>9</td>
</tr>
<tr>
<td>2.0 RELATED MICROPROCESSOR VERIFICATION EFFORTS</td>
<td>11</td>
</tr>
<tr>
<td>2.1 Tamarack</td>
<td>12</td>
</tr>
<tr>
<td>2.2 FM8501</td>
<td>12</td>
</tr>
<tr>
<td>2.3 VIPER</td>
<td>13</td>
</tr>
<tr>
<td>2.4 SECD</td>
<td>15</td>
</tr>
<tr>
<td>2.5 Comparison</td>
<td>15</td>
</tr>
<tr>
<td>3.0 THE FIVE-LEVEL STRUCTURE OF OUR VIPER IMPLEMENTATION</td>
<td>17</td>
</tr>
<tr>
<td>3.1 VIPER Instruction Level</td>
<td>18</td>
</tr>
<tr>
<td>3.2 The Macro Level</td>
<td>19</td>
</tr>
<tr>
<td>3.3 Micro Level</td>
<td>21</td>
</tr>
<tr>
<td>3.4 Phase Level</td>
<td>22</td>
</tr>
<tr>
<td>3.5 Electronic Block Level</td>
<td>23</td>
</tr>
<tr>
<td>3.5.1 The Data Path</td>
<td>23</td>
</tr>
<tr>
<td>3.5.2 The Control Unit</td>
<td>25</td>
</tr>
<tr>
<td>4.0 PROOF METHODOLOGY</td>
<td>29</td>
</tr>
<tr>
<td>4.1 Abstract operations</td>
<td>29</td>
</tr>
<tr>
<td>4.2 Verification Using an Abstract Interpreter Model</td>
<td>31</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>4.3 Hierarchical proof</td>
<td>33</td>
</tr>
<tr>
<td>5.0 MACRO LEVEL SPECIFICATION AND PROOF OF MICRO LEVEL</td>
<td>37</td>
</tr>
<tr>
<td>5.1 Instantiation of the interpreter</td>
<td>37</td>
</tr>
<tr>
<td>5.2 Example specification</td>
<td>40</td>
</tr>
<tr>
<td>5.3 Proof obligations and example proof</td>
<td>41</td>
</tr>
<tr>
<td>6.0 MICROCODE SPECIFICATION AND PROOF OF PHASE LEVEL</td>
<td>43</td>
</tr>
<tr>
<td>6.1 Instantiating the generic interpreter</td>
<td>43</td>
</tr>
<tr>
<td>6.2 Specification of microinstructions</td>
<td>43</td>
</tr>
<tr>
<td>6.3 Proof obligations</td>
<td>46</td>
</tr>
<tr>
<td>7.0 PHASE SPECIFICATION, BLOCK SPECIFICATION AND PROOF</td>
<td>47</td>
</tr>
<tr>
<td>7.1 Description of the phases</td>
<td>47</td>
</tr>
<tr>
<td>7.2 Description of block level</td>
<td>52</td>
</tr>
<tr>
<td>7.3 Proof of the Block level</td>
<td>52</td>
</tr>
<tr>
<td>8.0 MACRO LEVEL CORRESPONDENCE TO RSRE SPECIFICATION</td>
<td>57</td>
</tr>
<tr>
<td>8.1 Introduction</td>
<td>57</td>
</tr>
<tr>
<td>8.2 Methodology</td>
<td>57</td>
</tr>
<tr>
<td>8.3 Defining the instructions</td>
<td>58</td>
</tr>
<tr>
<td>8.4 Proof of SHLB</td>
<td>61</td>
</tr>
<tr>
<td>8.5 Definition of the Decoder</td>
<td>66</td>
</tr>
<tr>
<td>9.0 CONCLUSIONS</td>
<td>67</td>
</tr>
</tbody>
</table>
# APPENDICES

<table>
<thead>
<tr>
<th>Appendix</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>DESCRIPTION OF HOL</td>
<td>73</td>
</tr>
<tr>
<td>B</td>
<td>INTERPRETER THEORY AND ABSTRACT FUNCTIONS</td>
<td>77</td>
</tr>
<tr>
<td>C</td>
<td>VIPER LEVEL SPECIFICATION</td>
<td>89</td>
</tr>
<tr>
<td>D</td>
<td>MACRO LEVEL SPECIFICATION</td>
<td>111</td>
</tr>
<tr>
<td>E</td>
<td>MICRO LEVEL SPECIFICATION</td>
<td>157</td>
</tr>
<tr>
<td>F</td>
<td>MICROCODE</td>
<td>231</td>
</tr>
<tr>
<td>G</td>
<td>SAMPLE MACRO TO MICRO LEVEL PROOF</td>
<td>237</td>
</tr>
<tr>
<td>H</td>
<td>PHASE LEVEL SPECIFICATION</td>
<td>247</td>
</tr>
<tr>
<td>I</td>
<td>ELECTRONIC BLOCK LEVEL</td>
<td>267</td>
</tr>
<tr>
<td>J</td>
<td>INSTRUCTION DECODER</td>
<td>291</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.2-1 A microprocessor specification can be decomposed hierarchically.</td>
<td>6</td>
</tr>
<tr>
<td>3.1-1 VIPER Instruction Format</td>
<td>18</td>
</tr>
<tr>
<td>3.3-1 Microinstruction sequence for SHLS</td>
<td>22</td>
</tr>
<tr>
<td>3.5-1 Electronic Block Model</td>
<td>24</td>
</tr>
<tr>
<td>3.5-2 Microinstruction Format</td>
<td>25</td>
</tr>
<tr>
<td>4.1-1 Abstract representation of operations</td>
<td>30</td>
</tr>
<tr>
<td>4.1-2 Using an abstract representation</td>
<td>31</td>
</tr>
<tr>
<td>4.2-1 Abstract representation of a processor</td>
<td>31</td>
</tr>
<tr>
<td>4.2-2 Specification of the interpreter</td>
<td>32</td>
</tr>
<tr>
<td>4.2-3 Implementation of the interpreter</td>
<td>32</td>
</tr>
<tr>
<td>4.2-4 Obligations of the interpreter model</td>
<td>33</td>
</tr>
<tr>
<td>4.2-5 Intermediate lemma in final proof</td>
<td>34</td>
</tr>
<tr>
<td>4.2-6 Correctness of the interpreter</td>
<td>34</td>
</tr>
<tr>
<td>5.1-1 Macro-level viewed as an interpreter</td>
<td>37</td>
</tr>
<tr>
<td>5.1-2 Macro-instruction list</td>
<td>38</td>
</tr>
<tr>
<td>5.1-3 State as viewed by macro-instructions</td>
<td>38</td>
</tr>
<tr>
<td>5.1-4 Obligation for macro-instructions</td>
<td>39</td>
</tr>
<tr>
<td>5.2-1 The write_reg function</td>
<td>40</td>
</tr>
<tr>
<td>5.2-2 Example macro-instruction</td>
<td>40</td>
</tr>
<tr>
<td>5.3-1 Function to generate goals</td>
<td>41</td>
</tr>
<tr>
<td>5.3-2 Proof of SHLB instruction</td>
<td>42</td>
</tr>
<tr>
<td>6.1-1 Micro level interpreter in terms of the generic interpreter</td>
<td>44</td>
</tr>
<tr>
<td>6.2-1 State as viewed by microinstructions</td>
<td>44</td>
</tr>
<tr>
<td>6.2-2 Example microcode</td>
<td>45</td>
</tr>
<tr>
<td>6.3-1 Correctness of microinstructions</td>
<td>45</td>
</tr>
<tr>
<td>6.3-2 Correctness of the micro level</td>
<td>46</td>
</tr>
</tbody>
</table>
7.1-1 State manipulated by phase and EBM levels ................................................. 48
7.1-2 Description of first phase ....................................................................... 48
7.1-3 Description of second phase ................................................................. 49
7.1-4 Third phase ...................................................................................... 50
7.1-5 Third phase, continuation .................................................................... 51
7.2-1 Register with enable input ..................................................................... 52
7.2-2 Data path ........................................................................................ 53
7.3-1 Instantiating generic interpreter at phase level ........................................... 54
7.3-2 Tactic for proving individual phases ........................................................ 54
7.3-3 Proof of correctness of phase level ....................................................... 55
8.2-1 VIPER's NEXT function ....................................................................... 59
8.2-2 Goal for the verification step ................................................................... 59
8.4-1 Goal for proof of SLLB ....................................................................... 61
8.4-2 Lemmas for cases of DSF .................................................................... 62
8.4-3 Tactics in proof of SLLB .................................................................... 62
8.4-4 Lemmas with properties of VIPER level .............................................. 63
8.4-5 Error cases in VIPER specification ..................................................... 65
8.4-6 Tactic used in proof of SLLB ............................................................... 65
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.5-1 Comparison of verified microprocessors</td>
<td>16</td>
</tr>
<tr>
<td>3.2-1 VIPER macroinstructions</td>
<td>20</td>
</tr>
<tr>
<td>3.2-2 Decoding operand fields</td>
<td>21</td>
</tr>
<tr>
<td>A-1 HOL Infix Operators</td>
<td>74</td>
</tr>
<tr>
<td>A-2 HOL Binders</td>
<td>75</td>
</tr>
<tr>
<td>A-3 HOL Type Operators</td>
<td>76</td>
</tr>
</tbody>
</table>
1.0 INTRODUCTION

Computers are being used with increasing frequency in areas where the correct implementation of the computer hardware is critical. These include:

- Safety-critical applications where the computer is directly involved in the control of systems that protect human life. A flight control system on an aircraft or the control system in a nuclear power plant are examples of this type of application.

- Security-critical applications where the computer is used to process information that is economically or politically sensitive. Many computers used in government or industry fall into this category to one degree or another.

- Mass-produced consumer goods where the computer is an integral part of the product and a mistake in the design or implementation could result in product recalls costing enormous amounts of money.

In these and other applications it is vital that the computer system be correct.

There are two complementary approaches to computer correctness: fault tolerance and fault exclusion. The former, usually achieved through designs with redundant computing elements, is most useful in handling dynamic faults occurring during system operation, due to component failure or other unexpected events. The latter is a static process intended to remove errors in design and implementation before the computer system is in service.

Testing is an example of a fault exclusion technique. Testing can be divided into two distinct kinds: implementational testing, which is used to verify that a physical device is fabricated correctly, and functional testing, which is used to verify that a design functions as the designer intended. Because it is impossible to exhaustively test a computer system, formal verification is an attractive alternative to functional testing.

Formal verification requires at least two descriptions of a system: one of its implementation and one of its specification. Correctness is shown by demonstrating through mathematical proof that the former implies the latter. Although verification can be carried out using pencil and paper, the detail associated with the verification of realistic systems would overwhelm even the most patient human prover. Moreover, humans, being fallible, are likely to accept erroneous proofs as theorems. An alternative is the use of theorem proving programs. Such mechanical theorem provers
range from proof generators that attempt to create a proof with minimal human assistance to proof checkers that check a human-created proof. We used the HOL (Cambridge Higher Order Logic) theorem prover for our work. HOL's style of proof is closer to that of a proof checker than a proof generator, but HOL can be programmed to also provide significant automation in the creation of proofs.

Although through verification a computer system can, in principle, be demonstrated to contain no design errors, verification cannot in practice be guaranteed to achieve such a goal. First of all, the specification might not represent what the user wants of the system; in other words, the creation of the specification from informal requirements can introduce errors. Second, what is being verified, the implementation, is an abstraction of the physical device that comprises the microprocessor; the physical device might not correspond to the implementation, possibly due to errors introduced in the fabrication process. Third, verification, even with the assistance of mechanical theorem provers, is difficult and extremely human intensive; it might be impossible to complete the verification of complex systems.

Verification methodology has held the promise of correct programs for many years. However, it has been mostly impractical for large programs. In recent years, there has been interest in microprocessor verification. Although large programs are beyond the capability of the current verification technology, the verification of commercial microprocessors should be realistic. Our justifications for being optimistic about microprocessor verification are as follows:

- The specification for a microprocessor is not difficult to produce, largely expressing the functional behavior of each instruction.
- The implementation for many microprocessors is conceptually straightforward, largely involving iterative structures (such as registers) and control logic to resolve the many different cases. The algorithms represented by the implementation, even for arithmetic, are usually extremely simple compared with those associated with programs.

However, the detail involved in microprocessor proofs rapidly becomes staggering. This was the experience of Avra Cohn in attempting to verify the VIPER microprocessor.
1.1 VIPER

VIPER was designed by RSRE (ref. 1) in the mid-1980's. Not intended by its designers to push the envelope of microprocessor design, VIPER was designed to be simple and verifiable. For example, VIPER does not contain a stack or (user and privileged) modes, nor does it support interrupts. The first was excluded because it invites a programming practice that can lead to runtime errors, and the third because it was thought to be a feature difficult to verify. We have not seen comments on the second, but we conjecture that VIPER would not be used in any applications requiring multitasking.

Of interest to us here, are the attempts to verify VIPER, in particular (ref. 2). The top-level specification defines the NEXT state as a function of the current state and the current instruction. The elements of the state are main memory, five registers, and a few status bits—abstracting away a large fraction of the state that comprises the implementation. The implementation, called the electronic block model is described in terms of logical blocks such as an ALU, registers, flip-flops, multiplexors, etc. Both the specification and the electronic block model were provided to Cohn by RSRE. The proof was to demonstrate that the electronic block model implies the specification; HOL was used in the proof process.

Cohn's work remains a significant contribution, having formalized the electronic block model in HOL and having developed a methodology and many lemmas that could be used to carry out the proof. However, the proof was not completed. As it progressed, it became clear that approximately 1 person-week was required to prove the implementation of each of the 122 cases in the specification.

The difficulty was due to a number of factors, including:

a. RSRE's specification is extremely unstructured; essentially it is almost totally non-orthogonal. Although not conceptually difficult, the specification is still long—three pages of HOL logic.

b. Although not particularly complicated as compared with state-of-the-art commercial microprocessors, the implementation is still quite long. It occupies approximately seven pages of HOL logic. If this were a program being verified, by all measures it would be of nontrivial length.

c. Further elaborating on (b), the jump in abstraction between the specification and the electronic block model is too large to be carried out in one step.
d. There is insufficient support in HOL for the kinds of low-level reasoning associated with words, bit strings, etc.

It is item (c) that is of particular concern to us. In starting out on our work, we conjectured that through intermediate abstractions the proof effort required for VIPER could be simplified to the point where it would be realistic. It is still necessary to verify the lowest level of abstraction, defined in seven pages of HOL logic, ultimately with respect to the highest level of abstraction, occupying three pages of specification representing 128 cases. However, if the next to the lowest level of abstraction has fewer cases, the lowest level will be easier to verify. Similarly, if the next-to-highest level of abstraction is shorter, it will be relatively easy to verify with respect to the specification. The handcrafting of levels of abstractions is what is needed to simplify the verification of complex systems. In creating these abstractions, there will be tradeoffs among the number of cases, the size of the abstraction’s specifications, and the jump in data abstraction between adjacent abstractions.

As discussed later, the specification of the electronic block model of our VIPER machine is simpler than that of Cohn’s, with respect to omitted details not pertinent to our proof. For example, we do not specify in detail the logic of the ALU; instead it is declared to perform one of 32 unspecified functions. This incompleteness, of course, appears at all levels, including the top level. As noted by Brock and Hunt (ref. 3) with respect to a similar but less glaring weakness in the RSRE specifications of VIPER, the top-level specification does not permit proofs of programs that depend on the semantics of these operations to be carried out. However, the incompleteness in the electronic block model is not relevant to the main purpose of our verification effort: to verify that the sequence of actions at the electronic block model assure (among many other things) that the correct ALU control lines are asserted with respect to the instructions under execution.

VIPER has many more features that make it suitable for use in safety-critical applications, but are not modeled at the top-level. These include input signals for resetting the machine, single-stepping it, forcing the machine into an error state and extending read/write cycles. Output signals are also provided to indicate the state of the STOP and B flags, and whether the machine is currently fetching or executing an instruction. VIPER also incorporates a time-out facility in its interaction with the memory.

Because these features are inconsequential to the top-level specification, however, they can safely be ignored in the block-level specification, i.e. the implementation. However, for the purpose of verification with respect to the top-level instructions, certain assumptions about the behavior of these signals must be made. For example, the reset signal is assumed to be false throughout
the execution of an instruction and the STOP flag is assumed to be false at the beginning of an instruction. In addition, a simple memory model in which memory responds in a fixed and known number of cycles is being assumed, although the design of VIPER supports more complex memory protocols.

1.2 ABSTRACTION.

Viewing a complex program as a hierarchy of abstractions is a well-known approach to simplifying the verification of such a system. Programming languages such as Ada provide syntactic units (i.e., modules) for defining abstractions; of course, it is the programmer's responsibility to create modules that will simplify the design and, if it is relevant, the verification.

To facilitate the use of abstraction in the design and verification of microprocessors, Windley (ref. 4) formalized the concept of interpreters.

1.2.1 HIERARCHICAL DECOMPOSITION.

As mentioned above, verification requires at least two formal descriptions of the computer system: one behavioral, \( B \), and one structural, \( S \). Verification consists of showing through formal proof techniques that

\[
S \Rightarrow B
\]

One need not be limited, of course, to one level of abstraction. Supposing that \( B_1 \) through \( B_n \) represent increasingly abstract specifications of the system's behavior, one could verify its correctness by proving

\[
S \Rightarrow B_1 \Rightarrow \ldots \Rightarrow B_n
\]

Figure 1.2-1 shows how this principle can be applied to the specification of a microprogrammed microprocessor. At the bottom of the hierarchy is the usual structural specification of the electronic block model.
This specification describes the computer's implementation—for our purpose, the connections among its various components. At the top is the behavioral specification corresponding to the programmer's model of the microprocessor. In between these are two additional abstraction levels: one for the microcode interpreter and one specifying the phase (or subcycle) behavior. Our VIPER design has two macro levels: the topmost is the RSRE specification and the next lower specifies an orthogonal instruction set containing 20 instructions.

Hierarchical decomposition plays an important role in the methodology for verifying microprocessors. The use of a hierarchical decomposition can lead to significant reductions in the amount of effort used to structure and complete a correctness proof.

1.2.2 GENERIC INTERPRETERS.

With one exception, each of the levels in the specification hierarchy shown in Figure 1.2-1 has the same structure. The bottom-level specification is a structural description, but the other specifications all share a common structure. Each of the abstract behavioral descriptions can be specified using an interpreter model. However, the level in our hierarchy that corresponds to the RSRE instruction set does not fit exactly our interpreter model.
Perhaps the most distinguishing feature of an interpreter is that it has a flat control structure. One of $n$ instructions is chosen based on the current state. The chosen instruction operates on the state and the cycle begins anew. There are a large number of interesting computer systems that have a flat control structure: microprocessors, operating systems, language interpreters, and editors are a few.

Since each of the behavioral descriptions in the specification hierarchy are similar, we would prefer to develop a general model of an interpreter and use this model in our specification rather than treating each level in the hierarchy separately.

As we will demonstrate, a generic interpreter specification consists of a number of parts: abstract state, instructions, selectors for instructions, mapping to next lower state, description of implementation, etc. To verify the instantiation of a generic interpreter involves the verification of obligations, the most difficult of which is that each instruction is correctly implemented.

1.3 WHAT WE HAVE ACCOMPLISHED VIS-A-VIS VIPER

Our goal was to show that through the use of the generic interpreter methodology a microprocessor as complex as VIPER could be verified. Since VIPER was not designed as a hierarchy of interpreters, the RSRE VIPER design could not be verified using this methodology. Hence, we designed a microprocessor that would realize the VIPER instruction set as specified by RSRE. The design is in terms of the five levels of abstraction, as follows:

a. The top level is, with a few minor simplifications, the RSRE specification. In the RSRE specification, all functions (with the exception of a few arithmetic functions) are defined; in our specification some functions (such as the comparison of two words) are uninterpreted. As indicated previously, the exact meaning of functions used to define the instructions is not relevant to a proof that shows that the appropriate ALU signals are asserted for each instruction, and operands are fetched from and stored to the specified locations.

b. The next level down is the macro level specification (the top level of figure 1.2-1), providing 20 instructions. This level, as opposed to the RSRE specification, represents the VIPER instruction set in terms of comparatively few instructions with orthogonal fields. It is emphasized that this level is equivalent in power to the RSRE specification, but of course having a different format the instructions of this level would not execute VIPER programs. It was necessary to demonstrate that this level realizes the RSRE specification at level (a).
c. The third level is the *micro* level, providing approximately 100 microinstructions. Each macro instruction is implemented as a linear (loop-free) sequence of a subset of the microinstructions. The microcode is in effect the data of this level.

d. The next level down is the *phase* level, which implements each micro-instruction in a sequence of 3 phases.

e. The lowest level is the *Electronic Block Model* level, which consists of the control structure and datapaths to implement each of the phases.

Our experience to date has convinced us that the generic methodology has simplified the proof effort by half, as compared with Cohn’s experience. Furthermore, the use of hierarchical abstractions has permitted us to divide up the proof. Most of the proof was accomplished by two Master’s students, each student verifying 2 levels.

As Cohn has noted, it is important to clearly state what has been and what has not been verified.

Our proof demonstrates that the Electronic Block Model we have designed implements the RSRE instruction set. It is important to note that the ALU is a component of the Electronic Block Model. But having just specifications for the ALU, and not an implementation, means that we are not verifying that the ALU, when stimulated with signals that are assumed to cause it to add two numbers, actually does carry out the add operation. Of course, we could carry out the verification down to the gate-level—and verify the ALU, decoders, flip-flops, registers—and the other components taken as primitives of the Electronic Block Model. Such proofs are within current verification capabilities and in fact have been performed routinely by many verification teams.

When all is said and done, our verification shows the following: For each instruction of the RSRE specifications, the Electronic Block Model causes the proper sequencing of actions to take place; the operands are fetched from the right place (registers or memory), the results are stored in the right place, and the right signals are asserted on the primitive functional units (such as the ALU). Since there are many ways the Electronic Block Model could sequence activities (most of them incorrect) what is verified is far from trivial.
1.4 NOTATION AND CONVENTIONS.

Our notation will be that of standard logic with a few extensions:

- Terms in the logic will be written in **typewriter** font.
- Conjunction, disjunction, negation, implication, universal quantification, existential quantification, and lambda abstraction use the usual symbols: $\land$, $\lor$, $\neg$, $\implies$, $\forall$, $\exists$, and $\lambda$ respectively.
- We use a conditional operator that is written $a \rightarrow b | c$, meaning "if $a$, then $b$, else $c$.”
- Definitions will be denoted with a pre-pended $\vdash_{\text{def}}$.
- Terms that have been formally proven in the logic will be pre-pended with $\vdash$.

Other notations and logical expressions will be explained as they are used.

1.5 CHAPTER SUMMARIES.

Chapter 2 compares VIPER to other microprocessors that have been verified. Our Macro level shows that VIPER can be viewed as a microprocessor with approximately 20 instructions—about the same as several other microprocessors that have been verified. However, VIPER’s implementation complexity was reflected in the size of its microcode, i.e. approximately three times the complexity of other microprocessors considered for verification. The additional complexity is mostly due to error conditions.

Chapter 3 presents our design for the VIPER microprocessor, with the discussion organized according to the five levels of interpreters identified.

Chapter 4 reviews the hierarchical methodology employed in the verification. Excluding the top and bottom levels, each level in the hierarchy is a generic interpreter, which is instantiated to include the instructions supported by the interpreter, a unique key assigned to each instruction, the state space of the interpreter and its implementing interpreter, a mapping between these state spaces, and a description of the implementation. Once instantiated, an interpreter can be verified—showing that the implementation implies the specification for each instruction in the specification.

Chapters 5, 6, 7, and 8 highlight the verification effort. We discuss the specifications for each of the five interpreters and present in detail the verification of the shift-left instruction through the five levels.
Chapter 9 presents our conclusions and recommendations for future work. Particularly relevant are the recommendations for providing additional automation in the HOL system and the need for faster theorem proving engines. Although VIPER is a significant challenge to the current verification technology, it is still a rather impoverished microprocessor. Of interest, then, is scalability of the verification we and others working on microprocessor verification are pursuing: the prospects for verifying designs that are more complex than VIPER by an order of magnitude.

The 10 appendices include a brief description of the HOL logic (Appendix A) and the HOL listings of the five interpreters and the ML code that constitutes the verification. We have included the complete listings to allow the dedicated reader the opportunity to check our proof, to improve it through the use of better tactics, to extend the design with new features, or to translate the specifications into a different logic.
2.0 RELATED MICROPROCESSOR VERIFICATION EFFORTS

There have been numerous efforts to verify microprocessors. Many of these have used the same implicit behavioral model. We will first describe this implicit model and then describe the microprocessor verifications that use it.

In general, the model uses a state transition system to describe the microprocessor. The microprocessor specification has four important parts:

a. A representation of the state, S. This representation varies depending on the verification system being used.

b. A set of state transition functions, J, denoting the behavior of the individual instructions of the microprocessor. Each of these functions takes the state defined in step (a) as an argument and returns the state updated in some meaningful way.

c. A selection function, N, that selects a function from the set J according to the current state.

d. A predicate, I, relating the state at time $t + 1$ to the state at time $t$ by means of $J$ and $N$.

In some cases, the individual state transition functions, $J$, and the selection function, $N$, are combined to form one large state transition function. Also, a functional specification would use a function for part (4) instead of a predicate. The specifications, however, are largely the same.

After the microprocessor has been specified, we can verify that a machine description, $M$, implements it by showing

$$\forall s \in S \ M(s) \Rightarrow I(s).$$

That is, $I$ has the same effect on the state, $s$, that $M$ does. This theorem is typically shown by case analysis on the instructions in $J$ by establishing the following lemma:

$$\forall j \in J \ M(s) \Rightarrow (\forall t: \text{time} \ C(j, s, t) \Rightarrow s(t + n_j) = j(s(t)))$$

where $C$ is a predicate expressing the conditions for instruction $j$'s selection, $s(t)$ is the state at time $t$, and $n_j$ is the number of cycles that it takes to execute $j$. This lemma says that if an instruction $j$ is selected, then applying $j$ to the current state yields the state that results by letting the implementing interpreter $M$ run for $n_j$ cycles. We call this lemma the instruction correctness lemma.

The remaining parts of this section describe microprocessor verifications where some variation of this general model was used.
2.1 TAMARACK

Tamarack is a small microcoded microprocessor that has been verified by Jeffrey Joyce at the University of Cambridge. Joyce has verified Tamarack to the transistor level using HOL and has fabricated an 8-bit version of the design in CMOS. In addition to verifying the microprocessor, Joyce has also verified a compiler for Tamarack (ref. 5).

Tamarack is a 16-bit computer with a 13-bit address space. The computer has 8 instructions: halt, jump, jump if zero, add, subtract, load, store, and skip (or no operation). The architecture has an accumulator and a program counter visible to the assembly language programmer in addition to the memory. The computer is implemented in microcode and has a single bus connecting each of the blocks in the electronic block model. The microstore is 32 microwords long.

Tamarack is based on a computer designed and verified using the LCF-LSM system (a precursor to HOL) by Mike Gordon (ref. 6). Daniel Weise verified Gordon's design using a Lisp-based system called Silica Pithecus (ref. 7) and Harry Barrow verified it using a system called VERIFY (ref. 8), making this the most widely verified microcomputer design.

The specification and verification of Tamarack corresponds closely to the general model developed at the beginning of this section. The macro-level specification denotes what each instruction does and ties the descriptions of each instruction together with a predicate stating the relation between the state at time $t$ and time $t + 1$.

The verification of Tamarack is enlightening since it has been performed many times with many different verification systems and using many levels of abstraction. Tamarack is, however, small, and research is underway to discover methods for scaling the Tamarack experience to larger microprocessors, including those with larger instruction sets and support for operating systems.

2.2 FM8501.

FM8501 is a microprocessor designed and verified by Warren Hunt using the Boyer-Moore theorem prover (ref. 9). The architecture has a register file containing eight, 16-bit registers, a 64K-byte memory space, 26 instructions, and four memory addressing modes. FM8501 models memory as an asynchronous process. The implementation is microcoded and has a microstore of 16 microwords.
The specification of FM8501 consists of two recursive functions: one for the behavioral specification and one for the implementation. The functions recurse at each clock cycle, computing a new state. Time and the asynchronous inputs to the CPU are modeled by an oracle. The oracle is represented by a list; it is this list that the specifications recurse on. Time is represented by the current position of the recursive specification in the list. Each member of the list gives whatever asynchronous inputs may exist at that time. The proof shows the equivalence of the two recursive functions using an abstract (uninterpreted) oracle function.

Crocker et al re-verified FM8501 using a specification written in ISPS in the SDVS verification system (ref. 10). The re-verification is significant because the work used no part of Hunt's work directly and thus represents an independent verification of the design using a different verification system.

On the surface, the verification of FM8501 appears quite different than the verification of Tamarack, but in fact, they are very similar. The methods of specification for the top-level can be seen as an instance of the general model presented at the beginning of this section. The verification, even though done on a functional specification in a first-order system, uses a form of the instruction correctness lemma to show that the electronic block model implements the top-level specification.

2.3 VIPER.

VIPER was designed by Britain's Royal Signals and Radar Establishment (RSRE) at Malvern to provide a formally verified microprocessor for use in safety-critical applications. VIPER's designer's chose not to include a stack and interrupts—anticipating that they might lead to difficulties in the verification. The machine was designed to halt on errors and raise an external exception. The fabrication was carried out by two separate manufacturers and is commercially available.

VIPER has a 20-bit program counter, a 32-bit general purpose accumulator, and two 32-bit index registers. VIPER has a single instruction format that allows the user to select a source register, one of four memory addressing modes, one of eight destinations, whether or not to compare, and one of sixteen ALU functions. In addition to the fields just mentioned, each instruction contains a 20-bit address. The VIPER design is described in detail in (ref. 1). The implementation is hardwired instead of being microcoded.
The combination of fields in the instruction format (excluding source and destination selections) yields 122 different instruction cases. Our analysis of the VIPER design (ref. 11) has characterized the VIPER instruction set using only 20 instructions. As we will see, this is an important distinction that bears on the difficulty of verifying VIPER, and motivated us to include a new macro level in our design.

VIPER is the first microprocessor intended for commercial use where formal verification was attempted. Again, the verification was not completed. While VIPER is significantly simpler than today's general purpose microprocessors, its verification provides a benchmark on the state-of-the-art in microprocessor verification.

The specification of VIPER attendant to previous proof efforts (by RSRE and others) is hierarchical, although the levels do not have the uniform structure of our specification. The top-level specification of VIPER developed by RSRE is similar in style to that of Tamarack (ref. 5). The next level of the specification is called the major-state machine and is a description of VIPER's major states. The next level in the specification is the electronic block model. The top two levels were specified first in LCF-LSM and later in HOL. The electronic block model was specified in HOL. Below the electronic block model the circuit was described using a hardware description language called ELLA and verified by "intelligent exhaustive simulation" (ref. 12).

A paper-and-pencil proof of correctness between the top-level of VIPER and the major-state machine was performed by RSRE. Because of the complexity of the lower-level (electronic block model to major state machine) proof, RSRE did not attempt a hand proof of this level. RSRE contracted with Avra Cohn at Cambridge University to formalize the top-level proof and perform the lower-level proof. Cohn describes her formal verification of the major-state machine with respect to the top-level specification in (ref. 13).

Cohn decided to forego the proof of the top-level correspondence in trying to verify the electronic block model since the major-state level specification and the electronic block model yielded dissimilar structures under cases analysis. Instead, she attempted to show a direct correspondence between the top-level and the electronic block model (ref. 14). Cohn's proof of this level remains incomplete because of the large case explosion that occurred and the size of the proofs in each of the cases. This is not to say that the proof could not be completed.

From Cohn's experience with VIPER, it seems clear that abstraction is critical in dealing with the large case explosion that occurs in these kinds of proofs. The major-state machine did provide a level of abstraction between the top-level and the electronic block model, but it appears to be
the wrong one. In addition, Cohn had almost no access to VIPER's designers and thus had little or no help in deciphering and understanding the mostly informal specification of the electronic block model.

2.4 SECD.

Brian Graham et al at the University of Calgary have undertaken the implementation and verification of the SECD machine (ref. 15). The SECD machine is an abstract Lisp machine invented by Landin to reduce lambda expressions (ref. 16). The variant of SECD implemented by Graham is described in (ref. 17). Graham's work is part of a larger effort at the University of Calgary to verify a complete system including a LispKit compiler as well as the SECD chip.

The architecture has four registers, called S, E, C, and D. The S register holds a stack pointer, the E register holds a pointer to the environment, the C register functions as a program counter, and D points to a stack used to dump the state of the machine. There are approximately 20 instructions and the implementation is microcoded.

The remarkable thing about the SECD proof is that even though the architecture is specialized, the specifications and proofs are done in a manner very similar to the proofs of the more conventional architectures described in the last three sections. The behavioral model corresponds to the general model described at the beginning of this section. The top-level specification is based on state-transitions and the description of the electronic block model is a predicate-based circuit description similar to both (ref. 5) and (ref. 14). The garbage-collection mechanism is implemented in hardware, and the proof was done without taking it into account. Work is in progress on a second proof that verifies the garbage-collection hardware and a second implementation.

2.5 COMPARISON.

Table 2.5-1 summarizes the designs of the four microprocessors presented in this section. The table, like all such tabulations, cannot hope to capture all of the important characteristics of the microprocessors, but the data presented does provide some basis for judging relative complexities.
<table>
<thead>
<tr>
<th></th>
<th>Tamarack</th>
<th>FM8501</th>
<th>VIPER</th>
<th>SECD</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Registers</td>
<td>2</td>
<td>8</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Instructions</td>
<td>8</td>
<td>26</td>
<td>20</td>
<td>21</td>
</tr>
<tr>
<td>Microcoded</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Microstore size</td>
<td>32 words</td>
<td>16 words</td>
<td>N/A</td>
<td>512 words</td>
</tr>
<tr>
<td>Interrupts</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Memory Model</td>
<td>async</td>
<td>async</td>
<td>sync</td>
<td>sync</td>
</tr>
<tr>
<td>Word Width</td>
<td>16-bit</td>
<td>16-bit</td>
<td>32-bit</td>
<td>32-bit</td>
</tr>
<tr>
<td>Memory Size</td>
<td>8K</td>
<td>64K</td>
<td>1M</td>
<td>16K</td>
</tr>
</tbody>
</table>

*Table 2.5-1: Comparison of verified microprocessors*
3.0 THE FIVE-LEVEL STRUCTURE OF OUR VIPER IMPLEMENTATION

The proof of correctness of the VIPER microprocessor requires that the formal description of VIPER's implementation (down to the Electronic Block Model - EBM) implies the formal description of VIPER’s high-level specification. Due to the complexity and expense of proving this directly, however, the original VIPER verification was never completed (ref. 18).

In order to simplify the proof effort so that it could be accomplished in a reasonable time, we described the specification and implementation of VIPER in the form of a hierarchy of abstract interpreters, as described in Chapter 1. Instead of directly relating the high-level specification and implementation descriptions, the high-level specification can be related to an intermediate and less-abstract interpreter, which can be related to a lower-level interpreter, and so on down to the implementation. Each lower-level interpreter can be said to implement the interpreter above it in the hierarchy. Although the number of theorems that must be proved increases, the theorems are typically simpler, and the overall proof effort is greatly reduced.

The following sections describe the architecture of each of the hierarchical levels and summarize the proof strategy used to verify VIPER. The hierarchical decomposition approach uses five levels:

a. VIPER instruction level—The RSRE specification. This is what the assembly-language programmer sees.

b. Macro Level—The high-level VIPER specification as an interpreter, it consists of 20 instructions.

c. Micro Level—The microcode level. Each high-level instruction is implemented by a series of microinstructions, which constitute the specification at this level.

d. Phase Level—This level decomposes the interpretation of a single microinstruction into the parallel execution of a set of elementary operations.

e. Electronic Block Level—The “implementation” level of the microprocessor, described in terms of blocks such as the registers and the ALU.

The following paragraphs describe each level.
3.1 VIPER INSTRUCTION LEVEL

VIPER's high-level architecture consists of three general-purpose 32-bit registers (called A, X and Y), a 20-bit program counter (called P), and a single-bit boolean register (B) that holds the results of comparison instructions. The registers X and Y are normally referred to as "index registers" because they are most commonly used for address indexing, although they can also be used as general purpose registers. There is also a STOP flag that is not accessible to a programmer, but indicates an error condition in the machine. Any illegal operation, arithmetic overflow or computation of an illegal address causes the STOP flag to be set.

A memory address is 20 bits, but the memory itself has 32-bit words. The address space is divided into a memory space and a peripheral space each addressed by 20 bits. The distinction between the two is made by an extra memory/I/O bit. Only the least significant 20 bits of the program counter are meaningful, and loading a '1' into any of the top 12 bits will cause the machine to halt (viz., the STOP flag becomes true).

An instruction word is 32 bits long and consists of an operation code in the most significant 12 bits plus a 20-bit address. The address field is also used as an offset or constant by some instructions. The opcode is further subdivided as shown in Figure 3.1-1.

The opcode subfields are not orthogonal and are interdependent in an intricate way. Briefly, these fields have the following function:

rf: A 2-bit source register selector for the computation (A, X, Y or P).

mf: A 2-bit memory address control field that indicates the mode of fetching the operand from memory (literal addressing, content addressing or offset addressing (offset X or Y)).

df: A 3-bit destination selector for an ALU computation (registers, memory space or I/O space).
cf: A 1-bit flag that indicates whether or not the instruction is a comparison.

ff: A 4-bit function selector to indicate which comparison (if instruction is a comparison) or which computation is to be done by the ALU.

The specifications for this level are given in Appendix C.

3.2 THE MACRO LEVEL

Although the 12 opcode bits allow 4096 possible instructions, many of the combinations have redundant subfields, or represent impossible conditions, so that there are only 122 unique possibilities. We have split the 122 cases into 20 instructions. The operations that are supported by these instructions fall into six categories: shifts, comparisons, arithmetic and logical operations, procedure calls, memory read/writes and input/output instructions. The complete instruction set is listed in Table 3.2-1, with the meaning of the operand fields explained in Table 3.2-2. The HOL definitions for the entire macro-level are in Appendix D.

The SHLS instruction is one of 20 instructions in our macro level. If the stop field is set, there is no state change. The new value for the program counter is computed by adding 1 to the current value. If the address is invalid, the stop field is set. Otherwise, the register to be shifted is determined, and the shift performed. Finally, the shifted result is written to the appropriate register and the overflow bit is set if appropriate.

The specification is described in more detail in Section 5.2. To verify that the macro-level realizes the VIPER instruction level it is necessary to map each of the 20 macroinstructions to the 12 opcode bits of the VIPER level. A decoder function is introduced that maps the 12 opcode bits into a 5 bit instruction field (for 20 instructions) and (nearly) orthogonal fields corresponding to source register select (2 bits), memory mode select (2 bits) and destination register select (2 bits). The comparison flag and function select fields of the VIPER instruction level are not needed at the macro level.
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operands</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOOP</td>
<td>dreg, sreg</td>
<td>No operation</td>
</tr>
<tr>
<td>SHRS</td>
<td>dreg, sreg</td>
<td>dreg := sreg shifted right (copy sign bit)</td>
</tr>
<tr>
<td>SHRBS</td>
<td>dreg, sreg</td>
<td>dreg := sreg shifted right through B</td>
</tr>
<tr>
<td>SHLS</td>
<td>dreg, sreg</td>
<td>dreg := sreg shifted left; STOP := overflow</td>
</tr>
<tr>
<td>SHLB</td>
<td>dreg, sreg</td>
<td>dreg := sreg shifted left through B</td>
</tr>
<tr>
<td>COMPARE</td>
<td>ff, sreg, m</td>
<td>compare sreg and m, depending on ff</td>
</tr>
<tr>
<td>ADDS</td>
<td>dreg, sreg, m</td>
<td>dreg := sreg + m; STOP := overflow</td>
</tr>
<tr>
<td>SUBB</td>
<td>dreg, sreg, m</td>
<td>dreg := sreg - m; B := borrow</td>
</tr>
<tr>
<td>SUBS</td>
<td>dreg, sreg, m</td>
<td>dreg := sreg - m; STOP := overflow</td>
</tr>
<tr>
<td>NEG</td>
<td>dreg, m</td>
<td>dreg := -m</td>
</tr>
<tr>
<td>ANDM</td>
<td>dreg, sreg, m</td>
<td>dreg := sreg AND m</td>
</tr>
<tr>
<td>NOR</td>
<td>dreg, sreg, m</td>
<td>dreg := sreg NOR m</td>
</tr>
<tr>
<td>XOR</td>
<td>dreg, sreg, m</td>
<td>dreg := sreg XOR m</td>
</tr>
<tr>
<td>ANDMBAR</td>
<td>dreg, sreg, m</td>
<td>dreg := sreg AND m-complement</td>
</tr>
<tr>
<td>CALL</td>
<td>m</td>
<td>Y := P; P := m</td>
</tr>
<tr>
<td>WRITEMEM</td>
<td>sreg, addr</td>
<td>mem[addr] := sreg</td>
</tr>
<tr>
<td>READMEM</td>
<td>dreg, mem</td>
<td>dreg := m (from memory space)</td>
</tr>
<tr>
<td>WRITEIO</td>
<td>sreg, addr</td>
<td>io[addr] := sreg</td>
</tr>
<tr>
<td>READIO</td>
<td>dreg, mem</td>
<td>dreg := m (from io space)</td>
</tr>
</tbody>
</table>

Table 3.2-1: VIPER macroinstructions
**sreg** = source register (one of A, X, Y, P)

**dreg** = destination register (one of A, X, Y, P)

**STOP** = flag which indicates machine has stopped

**B** = flag set by comparison operators and if overflow occurs

\[
\begin{align*}
  m &= \text{tail} & \text{if } mf=0 \\
   &\quad (\text{tail}) & \text{if } mf=1 \\
   &\quad (\text{tail+X}) & \text{if } mf=2 \\
   &\quad (\text{tail+Y}) & \text{if } mf=3 \\
  \text{addr} &= \text{tail} & \text{if } mf=1 \\
   &\quad \text{tail+X} & \text{if } mf=2 \\
   &\quad \text{tail+Y} & \text{if } mf=3 \\
\end{align*}
\]

*Table 3.2-2: Decoding operand fields*

### 3.3 MICRO LEVEL

Our proof of VIPER is based on a micro-coded design in order to be able specify VIPER as a hierarchy of interpreters using the paradigm described in (ref. 4). As a result, we are able to take advantage of the proof simplification afforded by this method.

Each macro level instruction is implemented by a series of microinstructions. The microcode execution traces for each macro instruction are presented in Appendix F. For example, the microinstruction trace for the SILLS instruction is illustrated in Figure 3.3-1.

The microprogram that implements the SHLS instruction uses 10 of the approximately 100 microinstructions supported by the micro level. Many instructions use the same microinstructions, e.g., for fetching instructions, incrementing the program counter, etc. The microinstruction AXY_WRITE assures that the destination register is one of a, x, y. For this instruction, the destination cannot be the program counter. The microinstruction SHLS_u1 performs the actual shift and the write to the destination register.

A symbolic description of the VIPER microinstructions and the specification of the entire micro level are given in Appendix E. The microinstruction format is described in Section 3.5.2.
<table>
<thead>
<tr>
<th>Cycle</th>
<th>uCode</th>
<th>uLoc</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>fetch_u1</td>
<td>0</td>
<td>fetch macro instruction</td>
</tr>
<tr>
<td>t + 1</td>
<td>fetch_u2</td>
<td>1</td>
<td>increment pc</td>
</tr>
<tr>
<td>t + 2</td>
<td>fetch_u3</td>
<td>2</td>
<td>invalid address (&gt; 20 bits)?</td>
</tr>
<tr>
<td>t + 3</td>
<td>fetch_u4</td>
<td>3</td>
<td>ir ← macro instruction</td>
</tr>
<tr>
<td>t + 4</td>
<td>jmp_reqm</td>
<td>4</td>
<td>require memory?</td>
</tr>
<tr>
<td>t + 5</td>
<td>jmp_opc</td>
<td>5</td>
<td>jump to noop+instruction number</td>
</tr>
<tr>
<td>t + 6</td>
<td>AX_Y_WRITE</td>
<td>10</td>
<td>destination must be register A, X or Y</td>
</tr>
<tr>
<td>t + 7</td>
<td>SHLS_u1</td>
<td>11</td>
<td>shls operation</td>
</tr>
<tr>
<td>t + 8</td>
<td>NO_OVL</td>
<td>12</td>
<td>result must not overflow</td>
</tr>
<tr>
<td>t + 9</td>
<td>NOOP</td>
<td>13</td>
<td>jump to fetch next macro instruction</td>
</tr>
</tbody>
</table>

*Figure 3.3.1: Microinstruction sequence for SHLS*

### 3.4 PHASE LEVEL

The phase level, although it is the lowest level interpreter in the hierarchy, is more properly considered to be equivalent to the EBM level, rather than an abstraction of it. In particular, the phase and EBM levels share the same state and clock. Each phase in the system clock is associated with an instruction in the phase-level interpreter. The inputs to the phase-level interpreter consist of a bit-translation of the microinstructions defined for the micro level. In this way, the phase-level interpreter implements the micro-level interpreter.

Each microcycle (the time it takes to complete a single microinstruction) is composed of three phase cycles. The specification for the phase level, in Appendix H, has a separate definition for each of the phase cycles. The events that occur during each phase are described in Section 3.5.2.

The result at the first of three phases can be described in a simple way. At this level the state consists of a list of general-purpose registers (including a, x, y, p and others), registers to hold temporary results, the current instruction, data in and data out to memory (or I/O), the memory, b and stop bits, the memory address register and a result register for the ALU, the microprogram counter, the microinstruction register, the micro-ROM contents, 2 latches, and phase bits (to indicate the current and next phases). If the stop bit is set there is no state change, except to indicate there is no next phase. Otherwise, the contents of the micro-ROM as defined by the microprogram are fetched and control proceeds to phase 2. The other phases are similar, but much more complex, due to the complexity of the steps performed.
3.5 ELECTRONIC BLOCK LEVEL

The Electronic Block Model of VIPER used in the proof differs from the original RSRE design in several ways. Unlike the original design, the block model is microcoded to enable the use of the hierarchical decomposition proof method. The external interface is also different from that of the RSRE design in that it does not include certain input and output signals that have no effect with regard to the top-level specification. These signals were also ignored in Cohn's proof effort (ref. 18). Our VIPER Electronic Block Model is shown in Figure 3.5-1 and the EBM specification is in Appendix I.

3.5.1 THE DATA PATH

The data path consists of the registers at the phase level in addition to a few others (M, ONE and INS) that are used as internal scratchpad registers. INS is the instruction register, M is a temporary register used in operand computation and ONE holds the numerical constant '1'. Each of the programmer-accessible registers can output its contents onto the internal bus labeled r and the other registers can output contents onto the m bus. The least-significant 20 bits of P and INS can also be output to the MAR input bus. These registers can be loaded with either the ALU result or the word fetched from memory (DIN).

The m and the r buses feed into a 32-bit ALU that performs functions depending on the values of alucl (the ALU control signal from each microinstruction), ff and the B flag. The overflow and result of an operation are fed into both the register block and the micro-sequencing logic unit, which sets the STOP flag when an invalid result is generated in some contexts.

To communicate with memory, there is a 20-bit memory address register MAR, and two 32-bit data registers DIN and DOUT. The MAR can be loaded in parallel with an ALU operation. The MAR and DIN registers are loaded only if the r signal is set, and DOUT is loaded only if the w signal is set.

The instruction decoder unit takes in 12 bits of opcode from the INS register and the B flag, and sets the STOP flag if the opcode is illegal. Otherwise, it generates a condensed opcode. It also generates a signal reqm that denotes whether or not the instruction requires computation of an operand. This information is used by the microcode for branching purposes.

The STOP flag is set by both the instruction decoder and the micro-sequencing logic units. This is due to the fact that the machine could halt for two reasons - illegal instruction format (static error cases) and illegal operations during instruction execution (dynamic error cases).
Microinstruction Format

A microinstruction is 31 bits long. Its format is as shown in Figure 3.5-2. The interpretation of the microinstruction fields is given below.
maddr: address in the microcode, 7 bits

seqctl: 3 control lines for the micro sequencing logic:

(0 0 0) stay idle
(0 0 1) if reqm=true then mc := true; jaddr := maddr+mf[0..1]
(0 1 0) mc := true; jaddr := opc[0..4] + maddr
(0 1 1) mc := true; jaddr := maddr
(1 0 0) if overflow=true then stop:=true
(1 0 1) if (msb 12 bits of res has a 1) then stop := true
(1 1 0) if ((df[0..2]=3 or 4 or 5) V (msb 12 bits of res has a 1)) then stop := true
(1 1 1) if (df[0..2]=4 or 6) then stop:=true

aluctl: 4 control lines for the ALU, interpreted as:

(0 0 0 0) res := m
(0 0 0 1) res := r
(0 0 1 0) B := COMPARE(ff, r, m, b)
(0 0 1 1) res := -m
(0 1 0 0) res := r+m; B := carry
(0 1 0 1) res := r+m
(0 1 1 0) res := r-m; B := borrow
(0 1 1 1) res := r-m
(1 0 0 0) res := r XOR m
(1 0 0 1) res := r AND m
(1 0 1 0) res := r NOR m
(1 0 1 1) res := r AND NOT m
(1 1 0 0) res := r >> 1, copy sign bit
(1 1 0 1) res := r >> 1, shift through B
(1 1 1 0) res := r << 1, overflow := msb
(1 1 1 1) res := r << 1, shift through B

dec_ctl: control line to disable/enable the stop output of the instruction decoder

r: read signal
w: write signal
io: read/write from io (if true) or memory (if false)
mdf: destination select for alu result (for intermediate operations required by the instruction); 3 lines, decoded as:

(0 0 0) A
(0 0 1) X
(0 1 0) Y
mrf: source register select (for intermediate operations required by the instruction)

rfc: MUXR control line to decide which of rf/mrf is used to select source register

dfc: MUXD control line to decide which of df/dfc is used to select destination of alu result

dc: data enable, to enable data from memory to be written into reg block

de: res enable, to enable the ALU output to be written into reg block

adrs: address select, to select one of P/ADDR as the address

ds: data select, to select one of M/INS as destination of data from mem/io

Microinstruction Specification  A symbolic description of the VIPER microcode and the specific-ification for the micro level are given in Appendix E. As an example, consider the microinstruction number 19: SHLS.u2, the microinstruction that carries out the shift-left operation once the registers have been determined.

The state relevant to the microinstruction is that of the micro level, in particular the list of general purpose registers, the temporary (m), instruction, data input and data output registers, the memory, the overflow and stop bits, the memory address register, the (ALU) result register, the microprogram counter, and the reset bit. The RSF field determines the source field—the register whose contents are to be shifted. Assuming the stop bit is not set, the register determined by the DSF field receives the shifted contents of the source register, and the microprogram counter is incremented. All other state variables are unaffected.
**Microinstruction Timing**  Each microcycle is composed of three phase cycles, and the net effect of a microinstruction is an accumulation of effects of the three phases in sequence. Briefly, the events during each of the phases are as follows:

a. **Load the next microinstruction** to be executed into the microinstruction register MIR.

b. Gate the register values into the MLATCH, RLATCH. Load MAR with P or ADDR if r (read signal) is true. Load DOUT if w (write signal) is true. Set the STOP flag if either of the two stop conditions is true.

c. Load DIN with the value from memory if the read signal is true. Load the ALU result of data from memory into the register block. Load MPC with the address of the next microinstruction. Load RES and OVL with the ALU result and ALU overflow, respectively.

**Microinstruction Sequencing**  The address of the next microinstruction is either MPC + 1 or jaddr, which is computed by the micro-sequencing logic depending on all its inputs. The manner in which it is computed is given in the previous section as the explanation of the seqctl field of a microinstruction.
4.0 PROOF METHODOLOGY

The basis of this verification is the use of a package in HOL for abstract representation of functions and also a generic model for interpreters based on Windley's thesis. These two methodologies provide a way to separate critical control aspects from implementation-level details of concrete data operations. Each of these applications of abstract representations is explained before describing details of the verification of our VIPER design.

4.1 ABSTRACT OPERATIONS

The primitive functions performed by the machine and used in the specification of higher-level actions are defined as abstract operations. The HOL specification of these operations is shown in Figure 4.1-1. In particular, one may note that the operations are typed using type variables instead of concrete types (i.e. wordn instead of wordn).

Abstract functions are packaged together into abstract representations, which makes such "definitions" possible. Each abstract function can only appear once in any one theory, and the abstract representation can be accessed through the name of any of the functions defined in it. The type rep_ty given in Figure 4.1-2 is populated by all instances of the abstract representation defined in Figure 4.1-1. Any one function in an abstract representation can be used to key into a particular set of functions; in this case the function opcode is defined in Figure 4.1-1 and is used as a key in Figure 4.1-2. The universally quantified variable rep represents all possible instantiations for the set of abstract functions. The abstraction structure then becomes a parameter for all the other specifications that depend on these functions.

In our work, the functions of this abstract structure are given no meaning other than that illustrated in Figure 4.1-1. For example, all we say about add is that it maps two wordn's into a wordn. At all levels of the hierarchy add has only this meaning. Although not relevant to our proof, the exact meaning of add could be specified and shown to be correctly realized by an implementation of the ALU. This definition of add is reflected up to the instruction-level specification, and then assembly language programs referring to add could be verified.
new_theory 'aux_def';;

...........

let abs_rep = new_abstract_representation [
% ALU functions %
% negation %
('neg', ":(*wordn -> *wordn) ")

% addition without carry %
('add', ":(*wordn # *wordn -> *wordn) ")
;

% SHIFTER functions %

% shift left through b %
('shlb', ":(*wordn # bool -> *wordn) ")
;
% Coercion functions %
% numeric value of n-bit word %
('val', ":(*wordn -> num) ")
;

% Test functions %
% see if address is valid %
('valid_address', ":(*wordn -> bool) ")
;
% decoder %
('decode', ":((*opcode # bool) -> (bool # bt5 # bool)) ")
;

% Subranging functions %
% opcode portion of word %
('opcode', ":(*wordn -> *opcode) ")
;

% Memory functions %
% fetch a word from memory %
('fetch', ":((*memory # *address) -> *wordn) ")
;

];;

close_theory();;

Figure 4.1-1: Abstract representation of operations
let rep_ty = abstract_type 'aux_def' 'opcode';;

let load_m = new_definition('load_m', "! (rep:*rep_ty) (a:*wordn) (x:*wordn) (y:*wordn) (p:*wordn) (ir:*wordn) (ram:*memory).
load_m rep (a, x, y, p, ir, ram) = 
..........

4.2 VERIFICATION USING AN ABSTRACT INTERPRETER MODEL

The abstraction mechanism illustrated above is used not only to define the basic operations performed by the machine but also to model an "abstract" interpreter, or a general model for a processor that performs any given set of instructions. All the proofs of correctness of this abstract model of a processor are completed; thus all that is needed is to show that the specification and the implementation correspond to the same instantiation of the generic processor. These follow from the verification of a small set of proof obligations.

The components of an abstract interpreter are specified as shown in Figure 4.2-1. The complete specification is given in Appendix B. At any time, the pair (state, environment) selects a unique instruction to be executed next, through a given key. Each instruction provides a mapping from (state, environment) to state. The implementation (IMPL) is described as a predicate characterizing the state and environment values associated with the lower (implementation) level.

The abstraction specified by cpu_abs is used in the definition of two properties. INTERP, given
let I_rep_ty = abstract_type 'gen_I' 'key';;

let INTERP_def = new_definition
('INTERP',
""! (rep:'I_rep_ty) (s:time->*state) (e:time->*env).
INTERP rep s e =
  t:time.
    let n = (key rep (select rep (s t) (e t))) in (
      s(t+1) = (SND (EL n (inst_list rep))) (s t) (e t))"
);

Figure 4.2-2: Specification of the interpreter

let impl_imp_def = new definition
('IMPL_IMP',
""! inst:(*key#(*state->*env->*state))
(s':time'->*state')
(e':time'->*env') .
IMPL_IMP rep s' e' inst =
  (impl (rep:'I_rep_ty) s' e') =>
    (t:time'.
      let s = (\t. (substate rep (s' t))) in
      let e = (\t. (subenv rep (e' t))) in
      let c = (cycles rep (select rep (s t) (e t))) in ()
        (select rep (s t) (e t) = (FST inst)) /
        (count rep (s t) (e t) = (start rep)) =>
        ((SND inst) (s t) (e t) = (s (t + c))) /
        (count rep (s' (t + c)) (e' (t + c)) = (start rep)))"
);

Figure 4.2-3: Implementation of the interpreter

in Figure 4.2-2, denotes the fact that the state at the next cycle (s (t + 1)) must be the same as that specified by the instruction (SND (EL n (inst_list rep))), where the instruction itself is chosen by some function of the state and environment (select rep (s t) (e t)).

The other important property is represented by IMPL_IMP, shown in Figure 4.2-3. This function defines a function which, given the opcode of an instruction, asserts that if inst is the instruction currently selected, then after allowing the number of cycles necessary for the implementation to execute this instruction, the state is that specified by the instruction.

These two properties represent the semantics of an interpreter, one dealing with the state function and the other dealing with the meaning of each instruction. One step in the verification of a processor is to show that, if all the instructions are implemented correctly, then the next-state function's correctness follows. It is this step that is simplified by the use of the generic model.
To obtain the proof of correctness of the interpreter, one must first fulfill the necessary theory obligations, displayed in Figure 4.2-4.

The first of these theory obligations refers to a property to be maintained for each of the instructions. This property states that each instruction is implemented correctly. This is the most significant of the obligations as it is the most difficult to satisfy. The other two obligations relate to the ordering of the instructions, and to the fact that each opcode maps to a particular instruction.

Once all the proof obligations are discharged, the rest of the proof is completed automatically, by using the above properties as lemmas. For example, Figure 4.2-5 shows how a simplified version of IMPL_IMP is used in proving an intermediate lemma; in the code shown in Figure 4.2-6 we may observe how this lemma is used in the final proof of correctness of the processor— that the property INTERP (see Figure 4.2-2) holds at all times.

The use of the interpreter model thus becomes clear: the human verifier will "only" need to be concerned with the proof of each instruction and a few additional properties about the list structure of the instructions (the opcode); the interpreter model then combines all these proofs into a final proof of correctness for the processor.

### 4.3 Hierarchical Proof

Even when using the interpreter model to organize the proof effort, the verification of the RSRE VIPER micro-processor still involved a large number of cases to be verified, each of them quite complex. As explained previously, we have solved this problem by designing the architecture of the processor as a five-level hierarchy.
let IMPL_NEXTSTATE_LEMMA = TAC_PROOF
(([],
  "let s = (\t:time .(substate rep (s' t))) and
   e = (\t:time .(subenv rep (e' t))) in ( Impl (rep:"I_rep_ty)) s' e' ==> 
   (!t:time'.
   (count rep (s' t) (e' t) = (start rep)) ==> 
   ((substate rep (s' (t+(cycles rep (select rep (s t) (e t)))))) = 
   (SND (EL (key rep (select rep (s t) (e t)))
   (inst_list rep))) (s t) (e t)))"),
EXPAND LET_TAC
THEN REPEAT STRIP_TAC
THEN POP_ASSUM_LIST (\asl .
  let asl' =
    map (PURE_REWRITE_RULE [EVEKY_EL;IMPL_IMP_EXPANDED]) asl in
    MAP_EVERY ASSUME_TAC
THEN ............
THEN FIRST_ASSUM (ACCEPT_TAC o SYM_RULE)
);

Figure 4.2-5: Intermediate lemma in final proof

let IMPL_I_CORRECT = prove_thm
('IMPL_I_CORRECT',
  "let s = (\t:time .(substate rep (s' t))) and
   e = (\t:time .(subenv rep (e' t))) in ( Impl rep) s' e' /
   ((count (rep:"I_rep_ty)) (s' 0) (e' 0) = (start rep)) ==> 
   let f = time_shift (\st env. (cycles rep (select rep st env))) s e in
   (INTERP rep) (s o f) (e o f)",
EXPAND LET_TAC
THEN REPEAT GEN_TAC
THEN PURE_REWRITE_TAC [INTERP_DEF_EXPANDED;o_DEF]
THEN ............
THEN IMP_RES_TAC IMPL_NEXTSTATE_LEMMA_EXPANDED
);

Figure 4.2-6: Correctness of the interpreter

34
The interpreter model is used for all the proof levels. For example, at one level we consider the instantiation of the interpreter where the instruction list consists of the macro-instructions and the implementation is given by the micro-code. At another level, there is the instantiation with the instruction set being the micro-instructions and the implementation consisting of the phase-level description of the architecture.

The next sections describe proofs of the various levels in more detail. Each of these proofs consists of specifying the instruction set and the implementation, proving all the numerous lemmas—one for each instruction—that constitute the proof obligations, and then instantiating the proofs of correspondence for that level.

Chapter 5 presents the specification of the macro level (the second from the top in our five level hierarchy) in more detail than given in Chapter 3. Proof obligations are generated that relate to showing that the macro specification is correctly realized by the micro-level specification (including the microcode).

Chapter 6 presents the specification of the micro level and the proof that it is correctly realized by the phase-level specification.

Chapter 7 presents the specifications of the phase and electronic-block levels and proof of correspondence.

Finally, Chapter 8 presents the proof of the macro level with respect to the RSRE specification. We left this proof for last as the RSRE specification could not be conveniently captured in the generic interpreter theory.
5.0 MACRO LEVEL SPECIFICATION AND PROOF OF MICRO LEVEL

5.1 INSTANTIATION OF THE INTERPRETER

The macro-level view of VIPER is mapped to the interpreter model through the definition given in Figure 5.1-1.

The first parameter of INTERP is the set of macroinstructions \texttt{macro_inst_list}. The machine is specified by the action of 20 instructions, listed in Figure 5.1-2. The instruction \texttt{NOOP_M} is repeated so as to fill the opcode space up to 32 instructions. Each of these instructions is defined according to its effect on the \texttt{state} of the micro-level machine, defined in Figure 5.1-3. In the macro level, the processor state consists of the four data registers \texttt{a}, \texttt{x}, \texttt{y}, and \texttt{p}, the (overloaded) overflow flag register \texttt{b}, the \texttt{stop} signal, and the memory. Each instruction is specified as a function from a state to another state. The effect of an instruction on the state also depends on the \texttt{reset} signal, which is set by external processes and, thus, is not a part of the state under consideration.

Other parameters for instantiating the generic interpreter are:

- \texttt{Opcode} and \texttt{Opc_Val}: functions to select the macro-level opcode from the macro state and to instantiate the key, i.e. to index into the instruction list.

- \texttt{MacroLevelCycles}: a function that maps each instruction to the number of minor (i.e. micro) cycles necessary to complete the execution of the instructions; this number corresponds to the number of micro-instructions necessary to implement each macro-instruction.

- \texttt{Micro_state_to_Macro_state}: a function that indicates which parts of the micro-level state

```plaintext
let Macro_Int_def = new_definition
"! (rep:'rep_ty) (s:time->'macro_state) (e:time->'macro_env)
Macro_Int rep s e =
INTERP
(macro_inst_list rep, Opc_Val, Opcode rep,
MacroLevelCycles, Micro_state_to_Macro_state rep,
(I:'micro_env->'macro_env), Micro_I rep,
GetMPC, "FETCH_ADDR, @x:one.F)
s
e"
);
```

Figure 5.1-1: Macro-level viewed as an interpreter
let macro_inst_list = new_definition
('macro_inst_list',
"!! (rep: rep_ty).
macro_inst_list rep =
[((F,F,F,F,F),ABS_ENV (NOOP_M rep));
((F,F,F,F,T),ABS_ENV (SHR rep));
((F,F,F,T,F),ABS_ENV (SHRB rep));
((F,F,F,T,T),ABS_ENV (SHLB rep));
((F,F,T,F,F),ABS_ENV (SLH rep));
((F,F,T,F,T),ABS_ENV (CMP rep));
((F,F,T,T,F),ABS_ENV (WRITEM rep));
((F,F,T,T,T),ABS_ENV (WRITEM rep));
((F,T,F,F,F),ABS_ENV (NEG rep));
((F,T,F,F,T),ABS_ENV (CALL rep));
((F,T,F,T,F),ABS_ENV (READQ rep));
((F,T,F,T,T),ABS_ENV (READM rep));
((F,T,T,F,F),ABS_ENV (ADDQ rep));
((F,T,T,F,T),ABS_ENV (ADDS rep));
((F,T,T,T,F),ABS_ENV (SUBQ rep));
((F,T,T,T,T),ABS_ENV (NORM rep));
((T,F,F,F,F),ABS_ENV (XOR rep));
((T,F,F,F,T),ABS_ENV (AND rep));
((T,F,F,T,F),ABS_ENV (NORB rep));
((T,F,F,T,T),ABS_ENV (ANDM rep));
((T,F,T,F,F),ABS_ENV (NOOP_M rep));
((T,T,T,T,F),ABS_ENV (NOOP_M rep));
((T,T,T,T,T),ABS_ENV (NOOP_M rep));
((T,T,T,T,T),ABS_ENV (NOOP_M rep));

Figure 5.1-2: Macro-instruction list

let macro_state =
":*wordn#*wordn#*wordn#*wordn#bool#bool#memory"
% a x y p b stop ram %

let macro_env = ":(bool)";

Figure 5.1-3: State as viewed by macro-instructions
let Macro_Int_IMPL_IMPL_DEF = new_definition
('Macro_Int_IMPL_IMPL_DEF',
"!' (rep:'rep_ty) s' e'.
Macro_Int_IMPL_IMP rep s' e' =
IMPL_IMP
(macro_inst_list rep,
Opc_Val, Opcode rep, MacroLevelCycles,
Micro_state_to_Macro_state rep, (I:'micro_env->'macro_env),
Micro_I rep,
GetMPC, "FETCH_ADDR, ©x:one.F) s' e'"
));

Figure 5.1-4: Obligation for macro-instructions

are visible at the micro-level.

• I: the identity function, which signifies that the environment visible to the macro-level is identical to the one visible to the micro-level.

• Micro_I: the implementation, which is the (micro level) interpreter which executes the microcode, shown in Figure 6.1-1.

• GetMPC: a function that selects the micro-program counter from the state—the variable at the micro level that holds the current microinstruction.

• the start address: the opcode that signals the beginning of every micro-level execution.

Comparing these parameters to the abstract parameters used in the specification of the abstract interpreter illustrated in Figure 4.2-1 provides an illustration of how the abstraction mechanism works.

Once we have an instantiation of the generic interpreter, the next step is to satisfy the proof obligations, the heart of which is to prove that each macro-instruction is implemented correctly by the corresponding sequence of micro-instructions. In Figure 5.1-4 we can observe the instantiation of the function IMPL_IMP (see Figure 4.2-3) for this interpreter; note that even though the opcode does not appear in this instantiation, IMPL_IMP is a function that takes an extra numerical argument.
5.2 EXAMPLE SPECIFICATION

The macro-instructions are specified in terms of auxiliary functions, one of which is shown in Figure 5.2-1. The write_reg function defines which destination register is selected based on the DSF field.

The machine instruction for “shift left using the b register” is specified as shown in Figure 5.2-2. Given a particular state, the definition characterizes the state after the instruction is executed. The machine can already be in a stop state, in which case it will continue to be in that state. It will reach a stop state if the address of the next instruction (obtained by incrementing the program counter) is illegal. In all other cases, the machine will compute the result of applying the shlb abstract function to the contents of ldr, storing the result in the appropriate register and storing the bit shifted out into the b register.
let MK_INST_CORRECT_GOAL n =  
    let inst = term_list_el n  
    (snd(dest_eq(  
        snd(dest_forall(concl macro_inst_list)))) in  
        "!(rep:'rep_Zy) (regs:time->(*worda)list)  
            (m ins din dout:time->*wordn) (ram:time->*memory)  
            (b stop ov1:time->bool) (mar:time->*address)  
            (REG_LIST_LENGTH rep /
                DECODE_M_CORRECTLY_IMP rep) ==>
                (Macro_Int_IMPL_IMP rep  
                (\t. (reg t,m t,ins t,din t,douZ t, ram t,b t,stop t,  
                    ov1 t, mar t, res t, mpc t))  
                (\t. reset_e t) "inst");;

Figure 5.3-1: Function to generate goals

5.3 PROOF OBLIGATIONS AND EXAMPLE PROOF

In this section we describe the theorem that, when proved, asserts that the machine instructions are correctly implemented by the micro-code, and show how this theorem is proved. The proof consists primarily in showing that each of the 20 macro-instructions is implemented by its microprogram. The microcode appears in Appendix F.

An action to be repeated many times is the generation of goals: one for every macroinstruction. The goals are generated using the function given in Figure 5.3-1, repeatedly for each of the macroinstructions. (The argument for the function is the opcode: thus the function is iterated for all values from 0 to 19.)

The proof of the SHLB instruction is sketched in Figure 5.3-2. The opcode for SHLB is 3. The tactic FETCH_INST_TAC "simplifies" the goal by evaluating the results of fetching the instruction. Once the instruction is fetched and decoded, two cases arise: if the write to the destination register results in an exception condition then the machine stops; if not then the operation terminates successfully.

The proof may appear to be simple, but each of the tactics applied is very long and involved. FETCH_INST_TAC generalizes many steps needed in the proof:

- it specializes Macro_Int_IMPL_IMP_LEMMA to the appropriate macro-instruction,
- creates and proves the subgoal that the instruction has been decoded correctly,
- considers the number of cycles necessary for finishing each instruction,
- considers the case in which the machine is already in a stop state,
or goes into stop state due to an addressing exception.

The subgoal that remains is to prove the specific sequence of micro-instructions for the given instruction.

All the symbolic execution steps also involve manipulating the time aspects, and controlling the number of assumptions generated by resolution and rewriting tactics. These steps involve several layers of tactics, all of which are applied on each of the twenty goals (one for each instruction).

The proof for the other (19) instructions is similar to that of Figure 5.3-2. Each proof involves the tactic FETCH_INST_TAC and REWRITE_TAC, but tactics that deal with symbolic execution of the microcode and disposition of normal and error cases are a function of the instruction class in question. Thus, there are specialized tactics for addition, reading and writing memory, I/O, etc.
6.0 MICROCODE SPECIFICATION AND PROOF OF PHASE LEVEL

6.1 INSTANTIATING THE GENERIC INTERPRETER

The micro level of VlPER is also an instance of the generic interpreter, with the instruction list consisting of the microinstructions and the implementation being represented by the phase-level representation. The instantiation is given in Figure 6.1-1. It is useful to compare this instantiation with the one illustrated in Figure 5.1-1. The arguments of both are analogous.

6.2 SPECIFICATION OF MICROINSTRUCTIONS

The microinstructions operate on a more detailed state than the macro-instructions, as shown in Figure 6.2-1. Here, the four registers visible to the macro-instructions are modeled as a list of registers instead of a tuple. The other registers are: a temporary register m, the instruction register ir, and two memory data registers (for datain and dataout). Two boolean types represent the values of the b flag and the stop signal, while the other one is the internal overflow signal. The memory address register is of type address. The (temporary) value returned from the ALU is stored in the res register. The value of the microprogram counter is of type bt7. The reset signal is also visible.

The sequence of microinstructions needed to implement the SHLB macroinstruction is given in Appendix F: the first five cycles are used to fetch the macro-instruction, an optional memory fetch is performed (using up to seven additional cycles) and then four microinstructions specific to SHLB are executed.

One of the four microinstructions (SHLB_u2) called in the execution of SHLB is specified in Figure 6.2-2. This microinstruction, with opcode of 21, stores the value obtained by a left shift into the appropriate register, assuming the stop bit is not set.
let Micro_I_def = new_definition  
  ('Micro_I_def',  
    "! (rep:'rep_ty) (s:time->'micro_state) (e:time->'micro_env) .  
      Micro_I rep s e =  
      INTERP  
        (micro_inst_list rep,  
        bt7_val,  
        (GetMPC:'micro_state -> 'micro_env -> bt7),  
        (PhaseCycles:bt7->num),  
        (Phase_Substate:'Phase_state -> 'micro_state),  
        (I:'Phase_env ->'micro_env),  
        Phase_I rep,  
        (GetPhaseClock:'Phase_state -> 'Phase_env -> triple),  
        PhaseClockBegin, Ox:one.F) s e"  
  );;

let Micro_I_IMPL_IMPL_DEF = new_definition  
  ('Micro_I_IMPL_IMPL_DEF',  
    "! (rep:'rep_ty) (s:time->'Phase_state) (e:time->'Phase_env) .  
      Micro_I_IMPL_IMP rep s e =  
      IHPL_IHP  
        (micro_inst_list rep,  
        btT_val,  
        (GetMPC:'micro_state -> 'micro_env -> bt7),  
        (PhaseCycles:bt7->num),  
        (Phase_Substate:'Phase_state -> 'micro_state),  
        (I:'Phase_env ->'micro_env),  
        Phase_I rep,  
        (GetPhaseClock:'Phase_state -> 'Phase_env -> triple),  
        PhaseClockBegin, Ox:one.F) s e"  
  );;

Figure 6.1-1: Micro level interpreter in terms of the generic interpreter

let micro_state =  
  ":(((*wordn)list)#*wordn#*wordn#*wordn#*wordn#*memory  
  % a, x, y, p m ins din dout ram %  
  #bool#bool#bool#address#*wordn#bt7)";;  
  % b stop ovl mar res mpc %

let micro_env = ":(bool)";;  

Figure 6.2-1: State as viewed by microinstructions
let SHLB_u2 = new_definition
('SHLB_u2',
"!(rep:-rep_ty) (regs:(*wordn)list) (m ins din dout:*wordn) (ram:*memory)
(b stop ovl:bool) (mar:*address) (res:*wordn) (mpc:bt7)
(reset:bool).
SHLB_u2 rep (regs,m,ins,din,dout,ram,mar,res,mpc) (reset) =
let sval = shlb rep ((EL (bt2_val(KSF rep ins)) regs), b) in
stop => (regs,m,ins,din,dout,ram,b,T,ovl,mar,res,'FETCH_addr) |
(update_reg regs (DSF rep ins) sval, m, ins, din, dout, ram,
bitn rep (EL (bt2_val(RSF rep ins)) regs), F, F, mar, sval,
add_bt7 mpc 1)"
);

Figure 6.2-2: Example microcode

let PROVE_IMPL_IMP_LEMMA n =
TAC_PROOF (([],
MK_IMPL_IMP_GOAL n),
IMPL_IMP_TAC n));;

let MK_IMPL_IMP_GOAL n =
let inst = term_list_el n
(snd(dest_eq(
snd(dest_forall(concl micro_inst_list)))))) in
"!(rep:'rep_ty) (regs:(*wordn)list)
(mreg insreg din dout:(time->*wordn) (ram:(time->*memory)
(b stop ovl:(time->bool) (mar:(time->*address) (res:(time->*wordn)
(mpctime:(time->bt7)) (mir:(time->ucode) (rlatch mlatch:(time->*wordn)
(phi ph2 ph3:(time->bool) (reset:(time->bool)).
(t.
(stop t ==> "phi t \ 'ph2 t \ 'ph3 t) /
(phi t = "stop t \ 'ph2 t \ 'ph3 t) /
(ph2 t = "stop t \ 'phi t \ 'ph3 t) /
(ph3 t = "stop t \ 'phi t \ 'ph2 t) ==>
Micro_I_IMPL_IMP rep
(\t. (regs t, mreg t, insreg t, din t, dout t, ram t,
b t, stop t, ovl t, mar t, res t, mpc t, mir t, micro_rom,
rlatch t, mlatch t, phi t, ph2 t, ph3 t))
(\t. (reset t)) "inst";

let IMPL_IMP_TAC n =
let inst = term_list_el n
(snd(dest_eq(
snd(dest_forall(concl micro_inst_list)))))) in
let thm = el (n+1) instructions in
let find_Phase_I_term tm = (let ((x,y),z) = ((dest_comb # I)
(dest_comb tm)) in
(x = "Phase_I (rep:'rep_ty")") ? false in (REPEAT STRIP_TAC
THEN SUBST_TAC [SPEC inst Micro_IMPL_IMP_LEMMA]
THEN .......)
);

Figure 6.3-1: Correctness of microinstructions
let theorem_list =
  instantiate_abstract_theorems
  'gen_i'
  [Micro_I_CORRECT_LEMMA;
   Micro_I_LENGTH_LEMMA;
   Micro_I_ORDER_LEMMA]

("rep:'I_rep_ty",
 "(micro_inst_list (rep:'rep_ty),
 bt7_val,
 GetMPC:'micro_state->micro_env->bt7,
 Phase_Substate:'phase_state->'micro_state,
 (I:'phase_env->'micro_env),
 Phase_I rep,
 GetPhaseClock:'phase_state->phase_env->triple,
 PhaseClockBegin:triple,@x:one.F)"
);
("e':time'->*env'",
 "(\t:time. (reset t):bool)"
);
("s':time->*state'",
 "(\t:time. (reset t):bool)"
)
]
'MICRO';;

let correct_lemma = snd(hd theorem_list);;

let PHASE_IMPL_MICRO_LEMMA = save_thm
  ('PHASE_IMPL_MICRO_LEMMA',
   BETA_RULE (EXPAND_LET_RULE (ONCE_REWRITE_RULE [Phase_Substate;THM;GetPhaseClock;PhaseClockBegin] (BETA_RULE (ONCE_REWRITE_RULE [SYM_RULE Micro_I_def] correct_lemma)))))
);;

---

6.3 PROOF OBLIGATIONS

As in the proof of the macro level, the correct implementation of each of the microinstructions
must be proved. Here the number of lemmas needed is even larger than for the macro level—128
cases, corresponding to the 128 microinstructions—however all of them are appreciably simpler.
The process is repeated for each of the opcodes, as shown in Figure 6.3-1.

A single tactic (IMPL_IMP_TAC), when instantiated with the microinstruction number, suffices
to prove each of the 128 cases.

Once the proof obligations are met, the correctness lemma follows automatically. The proof,
where the lemmas and instantiations are used to obtain the final theorem of correctness for this
lemma, is shown in Figure 6.3-2.
7.0 PHASE SPECIFICATION, BLOCK SPECIFICATION AND PROOF

7.1 DESCRIPTION OF THE PHASES

Both the phase description and the block model manipulate the same state variables, given in Figure 7.1-1. Note the correspondence between this view and the structure represented in 3.5-1. Also note the variables introduced here (mir, urom, rlatch, etc.) not required in the micro-level specification.

The actions specified by each of the microinstructions are executed in three phases, each of which affect different subsets of the state variables. In the first phase the value of the microinstruction register is set by fetching the appropriate microinstruction from the micro-rom, as indicated by the value in the micro-program counter. This can be observed in the specification of **phase_one_def** given in Figure 7.1-2.

In the second phase, the micro-instruction is decoded. If the microinstruction calls for a 'read' or a 'write' operation the (source or destination) address is fetched into the mar. In the case of a 'write' the value to be written out is placed in dout. New values are also obtained for the two inputs for the ALU: rlatch and mlatch. The HOL definition for the second phase is given in Figure 7.1-3.

The destinations and other addresses are also checked for exceptions: in cases where any of the micro-operations are invalid, the stop signal is set and the processor does not execute the third phase; in other cases the machine is ready to run the third phase.

In the third phase the result computed by the ALU is stored in the appropriate register, and the address of the next microinstruction is computed and loaded into mpc, as shown in figures 7.1-4 and 7.1-5. The changes made during this phase are to the registers, the m register, the instruction register, the datain latch (in the case of a 'read' instruction), the memory in the case of a 'write', the flag b, the overflow indicator, the result from ALU, the mpc, and several others.

The three phases together, then, indicate the steps needed to execute a micro-instruction. Each of the 128 instructions takes three phases.
let Phase_state = ":(*wordn)list # % regs % (*wordn # % mreg % (*wordn # % insreg % (*wordn # % din % (*wordn # % dout % (*wordn # % ram % (*wordn # % b % (*wordn # % stop % (*wordn # % ovl % (*address # % mar % (*wordn # % res % (*wordn # % mpc % (bool # % mir % ((num -> ucode) # % rlatch % ((wordn # % mlnatch % (bool # % phase1 % (bool # bool))))))))))))))))") % phase2, phase3 %

let Phase_env = ":bool";;

Figure 7.1-1: State manipulated by phase and EBM levels

let phase_one_def = new_definition
(\'phase_one_def\',
phase_one rep (regs, mreg, insreg, din, dout, ram, b, stop, ovl, mar, res, mpc, mir, urom, rlatch, mlnatch, phi, ph2, ph3) (reset) =
    (regs, mreg, insreg, din, dout, ram, b, F, ovl, mar, res, mpc, urom (bt7_val mpc), urom, rlatch, mlnatch, F, T, F)\" ");;

Figure 7.1-2: Description of first phase

48
let phase_two_def = new_definition
  ('phase_two_def',   "\ (rep:*wordn)list (mreg insreg din dout:*wordn)
     (ram:*memory) (b stop ovl:bool) (mar:*address) (res:*wordn)
     (mpc:bt7) (mir:ucode) (urom:num->ucode) (rlatch mlatch:*wordn)
     (ph1 ph2 ph3:bool) (reset:bool).

phase_two rep (regs, mreg, insreg, din, dout, ram, b, stop, ovl,
  mar, res, mpc, mir, urom, rlatch, mlatch, phl, ph2,
  ph3) (reset) =
  (regs, mreg, insreg, din,
   %---- new dout ------%
   (W mir => EL (bt2_val(R_c mir => (Mrf mir)
       | RSF rep insreg)) regs
    | dout),
   ram, b,
   %---- new stop ------%
   ((FST(decode rep(opcode rep insreg,b)) \ (Dec_ctl mir))
    \ ((Seqctl mir = (F,F,T))
      \ ((FST(SND(decode rep(opcode rep insreg,b)))) = (F,F,T,T,F))
      \ ((FST(SND(decode rep(opcode rep insreg,b)))) = (F,F,T,T,T))
    \ (MSF rep insreg = (F,F))
    \ (Seqctl mir = T,F,F) \ ovl \ ...........
    \ (DSF rep insreg = (T,T,T))),
   ovl,
   %---- new mar ------%
   ((R mir \ W mir) => (Adrs mir => address rep insreg
     | address rep (EL p_reg regs))
    | mar),
   res, mpc, mir, urom,
   %---- new rlatch ------%
   EL (bt2_val (Rfc mir => (Mrf mir)
     | RSF rep insreg)) regs,
   %---- new mlatch ------%
   ((Ms mir = F,F) => mreg
    | ((Ms mir = F,T) => wordn rep 1
      | pad rep (address rep
        insreg)),
     F,F,
     %-- whether to go to phase three or not --%
     "((FST(decode rep(opcode rep insreg,b))
       \ (Dec_ctl mir)) \ .............
      \ (DSF rep insreg = (T,T,T))
    )") ;;

Figure 7.1-3: Description of second phase
let phase_three_def = new_definition
('phase_three_def',
"! (rep:'rep_ty) (regs:(*wordn)list) (mreg insreg din dout:*wordn)
  (ram:*memory) (b stop ovl:bool) (mar:*address) (res:*wordn)
  (mpc:btT) (mir:ucode) (urom:num->ucode) (rlatch m latch:*wordn)
phase_three rep(regs, mreg, insreg,din, dout, ram, b, stop, ovl, mar, res,
  mpc, mir, urom, rlatch, m latch, phi, ph2, ph3) (reset) =
((Re mir =>
  ((Dfc mir \ ((Mdf mir = (T,T,F)) \ (Mdf mir = (T,T,T)))) =>
    regs | update_reg regs
    (Dfc mir => (Mdf mir) | DSF rep insreg) b
    (((Aluctl mir = F,F,F,F) \ (Aluctl mir = F,F,T,F)) =>
      m latch |
    (((Aluctl mir = F,F,T,T) =>
      rlatch |
    ((Aluctl mir = F,T,F,F) \ (Aluctl mir = F,T,F,T)) =>
      add rep(rlatch,m latch) | ..........
      shi rep rlatch |
      shl rep rlatch | (shlb rep(rlatch,b))))))))))) |
  reg,),
(De mir =>
  (Ds mir => mreg | din) |
  ((Re mir \ Dfc mir \ ((bt3_val(Dfc mir =>(Mdf mir) | DSF rep insreg)=6)) =>
    .... ( (Aluctl mir = T,T,T,F) =>
      shl rep rlatch |
      shl rep rlatch | (shlb rep(rlatch,b))))))))))) |
  mreg) ),
(De mir =>
  (Ds mir => din | insreg) |
  ((Re mir \ Dfc mir \ ((bt3_val(Dfc mir =>(Mdf mir) | DSF rep insreg)=7)) =>
    join rep (opcode rep insreg, address rep
    (((Aluctl mir = F,F,F,F) \ (Aluctl mir = F,F,T,F)) =>
      m latch |
      ...... shi rep rlatch |
      shl rep rlatch | (shlb rep(rlatch,b))))))))))) |
  insreg ),
(R mir => (Io mir => fetchio rep(ram,mar) | fetch rep(ram,mar)) | din),
dout,
(W mir=>(Io mir=>storeio rep(ram,mar,dout) | store rep(ram,mar,dout)) | ram),

Figure 7.1-4: Third phase
(((Auctl mir = F,F,T,F) =>
  bcmp rep(rlatch,mlatch,b,FSF rep insreg) |
    .... ((Auctl mir = T,T,T,T) => bitn rep r latch | b))))),
F,
  aovfl rep(rlatch,mlatch,add rep(rlatch,mlatch)) |
  \(((Auctl mir = F,T,T,F) / (Auctl mir = F,T,T,T)) =>
  sovfl rep (rlatch,mlatch,sub rep(rlatch,mlatch)) |
  ((Auctl mir = T,T,T,F) => bitn rep r latch | F)));
mar,
  mlatch |
  ((Auctl mir = F,F,F,T) =>
    .... shl rep rlatch |
    shl rep(rlatch,b)))))))))),
...
bt7_ival((bt7_val mpc + 1)),
mir,urom,rlatch,mlatch,T,F,F)"

Figure 7.1-5: Third phase, continuation
7.2 DESCRIPTION OF BLOCK LEVEL

The block level is the lowest level of description in this verification, and consists of components such as the ALU, registers, flip-flops, etc. Proofs of each of the components are straightforward, although gate-level realizations can also be checked by testing. Small components, such as the register in Figure 7.2-1, are specified by their behavior. These are used in the structural specification of larger components such as the datapath, as shown in Figure 7.2-2. The components are linked by existentially quantified variables, which represent the internal lines of the implementation. This specification formalizes the block structure depicted in Figure 3.5-1.

7.3 PROOF OF THE BLOCK LEVEL

This proof also involves instantiating the generic interpreter model, as in the previous two levels. The instantiation is illustrated in Figure 7.3-1.

To establish the first theory obligation, we prove that Phase_I_IMPL_IMP applies to each of the three phases. The proof is relatively simple though it involves many rewrites and manipulation of long descriptions; the basic tactic used in all three proofs is shown in Figure 7.3-2.

The first obligation follows very easily from the proof of each of the lemmas. The other two obligations are also relatively straightforward, as we have to reason about a list of only three instructions. The proof is also made simpler because the two levels share the same clock, and they observe an identical state and environment. The final proof of correctness at this level is shown in Figure 7.3-3.
let DATAPATH = new_definition
  ('DATAPATH',
    "(! (rep:*rep_ty) (din dout rlatch mlatch res mreg insreg:time->*wordn)
         (b ovl reqm stop msl_stop ph2 ph3 rd wr io dfc din_en result_en
          addr_sel din_sel :time->bool)
         (mar:time->*address) (opc:time->bt5) (regs:time->(*wordn)list)
         (r_sel m_sel:time->bt2) (rft mft:bt2) (result_sel mdf:time->bt3)
         (dfc:bt3) (ram:time->*memory) (aluctrl:time->bt4)
         (dec_ctl reset:time->bool).
    DATAPATH rep din dout b mar rlatch mlatch res ovl reqm stop msl_stop
    ph2 ph3 regs mreg insreg rft mft dfc din_en result_en decCtl
    m_sel reset =
    "t:time.
    ? din_i mar_i rlatch_i mlatch_i result alu_ovl alu_b ir dec_stop.
    ((rft = RSF rep (insreg t)) /
     (mft = MSF rep (insreg t)) /
     (dfc = DSF rep (insreg t)) /
     (REGISTER_BLOCK rep result din ph3 r_sel result_sel din_en result_en
      addr_sel din_sel m_sel mar_i ir rlatch_i mlatch_i regs mreg insreg
      dcf mdf b) /
     (MAR_SPEC (\t. (rd t) \/(wr t))) ph2 mar_i mar) /
     (REG_EN_SPEC rd ph3 din_i din) /
     (REG_EN_SPEC wr ph2 rlatch_i dout) /
     (EXT_INTERFACE rep rd wr io ph3 mar dout din_i ram) /
     (REG_SPEC mlatch_i ph2 mlatch) /
     (REG_SPEC rlatch_i ph2 rlatch) /
     (alu_spec rep (rlatch t) (mlatch t) (result t) (alu_ovl t) (b t)
      (alu_b t) (aluctrl t) (FSF rep (insreg t))) /
     (REG_SPEC result ph3 res) /
     (FF_SPEC alu_ovl ph3 ovl) /
     (FF_SPEC alu_b ph3 b) /
     (INSDEC_SPEC rep (ir t) (b t) (dec_ctl t) (dec_stop t) (opc t)
      (reqm t)) /
     (STOP_SPEC stop dec_stop msl_stop ph2))")
  );;

Figure 7.2-2: Data path
let Phase_I_def = new_definition
('Phase_I_def',
"! (rep:rep_ty) (s:time->Phase_state) (e:time->Phase_env).
Phase_I rep s e =
INTERP
([ONE,phase_one rep; 
  TWO,phase_two rep; 
  THREE,phase_three rep], 
triple_value, 
(GetPhaseClock:Phase_state -> Phase_env -> triple), 
(PhaseLevelCycles:triple->num), 
(I:EBM_state->Phase_state), 
(I:EBM_env->Phase_env), EBM rep, 
(GetEBMClock:EBM_state->EBM_env->bool), 
EBM_Start, @x:one.F) s e")
);

let Phase_I_IMPL_IMP_DEF = new_definition
('Phase_I_IMPL_IMP_DEF',
"! (rep:rep_ty) s' e'.
Phase_I_IMPL_IMP rep s' e' =
IMPL_IMP
([ONE,phase_one rep; 
  TWO,phase_two rep; 
  THREE,phase_three rep], 
triple_value, 
(GetPhaseClock:Phase_state -> Phase_env -> triple), 
(PhaseLevelCycles:triple->num), 
(I:EBM_state->Phase_state), 
(I:EBM_env->Phase_env), EBM rep, 
(GetEBMClock:EBM_state->EBM_env->bool), 
EBM_Start, @x:one.F) s' e'"
);

Figure 7.3-1: Instantiating generic interpreter at phase level

let PHASE_EBM_TAC = 
PURE_ONCE_REWRITE_TAC [Phase_I_IMPL_IMP]
THEN REPEAT GEN_TAC
THEN BETA_TAC
THEN REWRITE_TAC [GetPhaseClock;PhaseLevelCycles; 
  GetEBMClock;EBM_Start;phase_one_def; 
  phase_two_def;phase_three_def]
THEN SUBST_TAC [EBM_expanded]
THEN REPEAT STRIP_TAC
THEN POP_ASSUM_LIST ((asl. (MAP_EVERY (STRIP_ASSUME_TAC o SPEC_ALL) asl)) 
THEN POP_ASSUM_LIST (\asl. (MAP_EVERY (STRIP_ASSUME_TAC o SPEC_ALL) asl));

Figure 7.3-2: Tactic for proving individual phases
let theorem_list = instantiate_abstract_theorems
  "gen_I"
  [Phase_I_EVERY_LEMMA;
   Phase_I_LENGTH_LEMMA;
   Phase_I_KEY_LEMMA]
  ["rep':I_rep_ty",
   "([ONE,phase_one (rep:~rep_ty);
      TWO,phase_two rep;
      THREE,phase_three rep],
    triple_value, (GetPhaseClock:"Phase_state->"Phase_env->triple),
    PhaseLevelCycles, (I:"EBM_state->"Phase_state),
    (I:"EBM_env->"Phase_env),
    EBM rep, (GetEBMClock:"EBM_state->"EBM_env->bool), EBM_Start")];
("e':time'->*env'",
 "([t:time. (reset t)):zimB->-EBM_env]
 ("s':time'->*state'",
 "([t:time. (regs t, mreg t, insreg t, din t, dout t, ram t,
    b t, stop t, ovl t, mar t, res t, mpc t, mir t, urom,
    rlatch t, mlatch t, phi t,
    ph2 t, ph3 t)):time->"EBM_state")
 ];
 'PHASE';;

let EBM_IMPL_PHASE_LEMMA = save_thm
  ('EBM_IMPL_PHASE_LEMMA',
   (ONCE_REWRITE_RULE [I_o_ID] (EXPAND_LET_RULE
    (ONCE_REWRITE_RULE
     [GetEBMClock;EBM_Start;'I_THM;'TIME_SHIFT_DEGENERATE_LEMMA]
      (BETA_RULE
       (ONCE_REWRITE_RULE [SYM_RULE Phase_I_def] correct_lemma)))));

Figure 7.3.3: Proof of correctness of phase level

55
8.0 MACRO LEVEL CORRESPONDENCE TO RSRE SPECIFICATION

8.1 INTRODUCTION

This section describes the verification of our macro level with respect to the level that defines the VIPER instructions. The VIPER instruction level, as specified by RSRE, is not in the format of our generic interpreter. Hence we are employing a style of proof here different from that used in the other levels.

In general terms, the verification described in this section involves showing that each possible opcode in the VIPER level is realized by one of the 20 instructions at the macro level with suitable values for the three fields: source register select, destination register select, and memory mode select. The opcode is a 12-bit field, thus there are $2^{12}$ different values possible for the opcode. An abstract decoder is assumed, which maps the 12 opcode bits of the VIPER level to an instruction and to the three selection fields at the macro level.

The VIPER level is divided up into cases, each of which (with a few exceptions) corresponds to one of the 20 macroinstructions. Then it is shown that these cases cover the $2^{12}$ possible values for the VIPER-level opcode fields.

8.2 METHODOLOGY

The NEXT function, as shown in Figure 8.2-1, is the heart of the VIPER instruction specification. The NEXT definition is primarily a decoding tree, which determines the subsequent state based on the current values in the VIPER registers and memory. For instance, if the 'comp' flag is set, the machine will execute a compare operation. If a write operation is requested, VIPER will attempt to execute a write operation.

Even though there are different fields in the instruction register, namely DSF, CSF, FSF, and MSF, the interpretations of these registers are not independent of each other. For example, MSF is usually used to decide which addressing mode the processor will use to access memory, unless FSF is (T,T,F,F), in which case MSF will be used to decide which shift operation the machine will execute. This lack of orthogonality complicates the verification with respect to the NEXT function because the verifier must “walk through” the decoding tree for each combination of DSF, CSF, and MSF and determine the behavior of each instruction. This lack of orthogonality complicated Cohn’s proof.
This definition cannot serve as the top level in the interpreter hierarchy, as we have defined interpreters. An orthogonal instruction set has to be derived and used as the macro level—the top level in the abstract interpreter hierarchy. Furthermore, to prove our implementation of VIPER, we also have to prove that the our macro level is equivalent to the NEXT state definition as defined by RSRE and used by Cohn.

The proof methodology is as follows. To define the top level in our hierarchy—the RSRE level—first we define an interpreter using the RSRE definition of the NEXT state function of Figure 8.2-1, referred to as cohn_NEXT:

\[
\begin{align*}
\text{cohn}_\text{Int} &\ 
\text{rep} \ (s, e) = \\
&\ \text{(! } t, s(t+1) = \text{cohn}_\text{NEXT} \ 
\text{rep} \ (s, t))
\end{align*}
\]

Then the goal to be proved is illustrated in Figure 8.2-2. It expresses the desired property that for all possible states visible at the macro and VIPER levels, characterized by combination of (a, x, y, p, b, stop, ram), the macro interpreter yields the same next state as the RSRE level characterized by the NEXT function.

To minimize the cases we have to consider, we start with a decoder for the interpreter. The decoder in the interpreter is responsible for determining from the state 7-tuple the correct instruction for the macro level. For each major case that the decoder generates, we define an instruction to handle that case. For instance, if the CSF bit is set, the decoder should select the CMP instruction— a bit compare. If the DSF field is (T, T, F) and the CSF bit is not set, the decoder should select the WRITEIO instruction. Thus the somewhat ill-structured VIPER instruction set is mapped to an orthogonal set. The cases for the decoder and the corresponding instructions are listed in Appendix J.

8.3 DEFINING THE INSTRUCTIONS

The macro-level instructions can be divided conveniently into five classes of instructions. The first class includes instructions that do not access memory. This class includes the shift instructions, of which there are four: SHR, SHL, SHRB, SHLB, for right and left shifts using or not using the b register. There are four cases for each shift instruction, corresponding to the four source registers (a, x, y, p), as specified by the DSF field. The load_r function performs this selection.
\[ \text{NEXT} \ (\text{ram}, \ p, \ a, \ x, \ y, \ b, \ \text{stop}) = \]

\[
\begin{align*}
\text{...} & \quad \text{...} \\
(\text{stop} \Rightarrow (\text{ram}, \ p, \ a, x, y, b, T) \in \\
((\text{noinc} \lor \text{illegaladdr}) \lor ((\text{illegalcl} \lor \text{illegalsp}) \\
\lor (\text{illegalonp} \lor \text{illegalwr}))) \Rightarrow \\
(\text{ram}, \ \text{newp}, \ a, x, y, b, T) \in \\
(\text{comp} \Rightarrow (\text{ram}, \ \text{newp}, \ a, x, y, \text{COMPARE}(\text{fsf}, \ \text{source}, \\
\text{MEMREAD}(\text{ram}, \ \text{msf}, \ \text{addr}, \ x, y, \text{io}, F), b), F) \lor \\
(\text{writeop} \Rightarrow (\text{MEMWRITE}(\text{ram}, \ \text{source}, \ \text{msf}, \ \text{addr}, x, y, \text{io}), \ \\
\text{newp}, \ a, x, y, b, F) \lor \\
(\text{skip} \Rightarrow (\text{ram}, \ \text{newp}, \ a, x, y, b, F) \lor \\
\text{let } m = \text{MEMREAD}(\text{ram}, \ \text{msf}, \ \text{addr}, x, y, \text{io}, \text{WILM(dsfs, csfs, fsfs)}) \text{ in} \\
\text{let } \text{aluout} = \text{ALU}(\text{fsfs}, \ \text{msfs}, \ \text{dsfs}, \ \text{source}, m, b) \text{ in} \\
((\text{df} = 0) \Rightarrow (\text{ram}, \ \text{newp}, \ \text{VALUE} \ \text{aluout}, x, y, \\
\text{BVAL} \ \text{aluout}, \ \text{SVAL} \ \text{aluout}) \lor \\
((\text{df} = 1) \Rightarrow (\text{ram}, \ \text{newp}, \ a, \ \text{VALUE} \ \text{aluout}, y, \\
\text{BVAL} \ \text{aluout}, \ \text{SVAL} \ \text{aluout}) \lor \\
((\text{df} = 2) \Rightarrow (\text{ram}, \ \text{newp}, \ a, x, \ \text{VALUE} \ \text{aluout}, y, \\
\text{BVAL} \ \text{aluout}, \ \text{SVAL} \ \text{aluout}) \lor \\
(\text{call} \Rightarrow (\text{ram}, \ \text{TRIM32TO20}(\text{VALUE} \ \text{aluout}), a, x, \\
\text{INCP32 p}, \ \text{BVAL} \ \text{aluout}, \ \text{SVAL} \ \text{aluout}) \lor \\
(\text{ram}, \ \text{TRIM32TO20}(\text{VALUE} \ \text{aluout}), a, x, y, \\
\text{BVAL} \ \text{aluout}, \ \text{SVAL} \ \text{aluout}))))))) & \\
\text{set_goal([],} & \\
"! (\text{rep}:'\text{rep}_\text{ty}) \ (\text{a}:\text{time}->\text{wordn}) \ (\text{x}:\text{time}->\text{wordn}) \ (\text{y}:\text{time}->\text{wordn}) \ (\text{p}:\text{time}->\text{wordn}) \ (\text{b}:\text{time}->\text{bool}) \ (\text{stop}:\text{time}->\text{bool}) \\
(\text{ram}:\text{time}->\text{memory}) \ (\text{t}:\text{time}) \cdot \\
(! (\text{ram}:'\text{memory}) \ (\text{p}:'\text{wordn}) \cdot \\
((\text{address rep (pad rep (address rep \\
(fetch rep (\text{ram}', \ \text{address rep p'}))))}) \\
= \text{address rep (fetch rep (\text{ram}', \ \text{address rep p'}))))} & \\
\Rightarrow \\
((\text{Macro_Int rep (\text{t}. \ ((\text{a t}), \ \text{x t}), \ \text{(y t)}, \ \text{(p t)}, \ \text{(b t)}, \ \text{(stop t)}, \\
(fetch rep (\text{ram t}), \ \text{address rep (p t)}), \\
\text{ram t})))) \ (\text{t}.(\text{reset t}))) = \\
(\text{cohn_Int rep (\text{t}. \ ((\text{a t}), \ \text{x t}), \ \text{(y t)}, \ \text{(p t)}, \ \text{(b t)}, \ \text{(stop t)}, \\
\text{fetch rep (\text{ram t}), \ text{address rep (p t)}), \\
\text{ram t})))) \ (\text{t}.(\text{reset t}))))");;
\end{align*}\]

Figure 8.2-1: VIPER's NEXT Function

Figure 8.2-2: Goal for the verification step
The second class of instructions are those that write to memory: \texttt{WRITEM} and \texttt{WRITEIO}. There are 16 subcases for the \texttt{WRITEM} instruction, corresponding to the possible selections of source and destination registers. The proof entails reasoning about 4 subcases for each of the 4 instructions.

The above two classes of instructions do not require any memory read. The third set of instructions are those that read memory, wherein the result cannot be used to modify the register. These instructions are: \texttt{ADDB}, \texttt{SUBB}, \texttt{NEG}, \texttt{XOR}, \texttt{AND}, \texttt{NOR}, \texttt{ANDMBAR}, and \texttt{READIO}.

There are four cases of memory load and six cases of output writes (three valid and three invalid) yielding a total of 24 subcases for each of these instructions. The memory reads can be generalized so there are only six subcases to be proved for each such instruction.

The fourth set of instructions are similar to the third set but they involve writing to the register, in effect achieving a jump or a goto. The specific instructions are: \texttt{CALL}, \texttt{READM}, \texttt{ADDS}, and \texttt{SUBO}.

The specification for \texttt{CALL} is basically the same as the \texttt{ADDB} instruction except for some minor difference in \texttt{write.preg}. Similar to the \texttt{ADDB} instruction, there are six subcases for each of these instructions.

The last class of instruction is what we call the compare instruction. We have decided to have an abstract function \texttt{bcmp} representing all sixteen cases of compare. The \texttt{bcmp} function appears at all levels, including the block level. The memory load is generalized so there is only one case to prove for the compare instruction.
8.4 PROOF OF SHLB

As in the previous sections, we have chosen SHLB to illustrate the proof methodology. First we identify the conditions on the VIPER-level state under which SHLB is selected, namely:

\[
\neg \text{CSF} \land \\
\neg (\text{DSF} = (T, T, T) \lor \text{DSF} = (T, T, F)) \land \\
\neg (\text{DSF} = (T, F, T) \land b) \lor (\text{DSF} = (T, F, F) \land b) \\
\neg \text{FSF} = (F, F, F, T) \land \text{FSF} = (T, T, F, F) \land \text{MSF} = (T, T)
\]

Hence, the goal for the verification of SHLB can be written as in Figure 8.4-1. The goal states that if the conditions for invoking the SHLB instruction are satisfied then the effects of the SHLB instruction on the macro state are identical to those specified by the NEXT function. As a lemma we have proved that the register selected at the macro level and the VIPER level is the same:

\[
\neg \text{CSF} \\
\neg (\text{DSF} = (T, T, T) \lor \text{DSF} = (T, T, F)) \land \\
\neg (\text{DSF} = (T, F, T) \land b) \lor (\text{DSF} = (T, F, F) \land b) \\
\neg \text{FSF} = (F, F, F, T) \land \text{FSF} = (T, T, F, F) \land \text{MSF} = (T, T)
\]

We also have decomposed the NEXT definition into cases corresponding to each DSF value, as illustrated in Figure 8.4-2. Thus we can rewrite the NEXT definition much faster in our proof. There are six states for the DSF so six such theorems are required.

We now dispose of simple cases (e.g., stop, invalid new program counter after increment) by using the tactic illustrated in Figure 8.4-3.
(let fsf = (FSF rep (fetch rep (ram, address rep p))) in
let dsf = (DSF rep (fetch rep (ram, address rep p))) in
let msf = (MSF rep (fetch rep (ram, address rep p))) in
let rsf = (RSF rep (fetch rep (ram, address rep p))) in
let addr = (address rep (fetch rep (ram, address rep p))) in
let newp = (add rep (p, wordn rep 1)) in
let io = ((cohn_OUTPUT rep (dsf, csf)) V
(cohn_INPUT rep (dsf, csf, fsf))) in
let r = cohn_REG rep (rsf, a, x, y, newp) in
let m = cohn_MEMREAD rep (ram, msf, addr, x,
y, io, cohn_NILM rep (dsf, csf, fsf)) in
let aluout = cohn_ALU rep (fsf, msf, dsf, r, m, b) in
let newp = (add rep (p, wordn rep 1)) in
(((not (stop)) A
("csf") A
(valid_address rep newp) A
("(dsf = (T,T,T))") A
("(dsf = (T,T,F))") A
(dsf = (F,F,F)) A
(fsf = (T,T,F,F)) ==>
(cohn_NEXT rep (a, x, y, p, b, F, ram) =
(cohn_VALUE aluout, x,
y, newp,
cohn_BVAL aluout, cohn_SVAL aluout, ram))))
Next we step through the DSF cases by first considering DSF = (F, F, F). We must identify the values for (cohn\_VALUE aluout), (cohn\_BVAL aluout), and (cohn\_SVAL aluout). The theorems displayed in Figure 8.4-1 characterize the values required in the proof.

By specializing the above theorems, and under the condition that the goal preconditions hold and DSF = (F, F, F), we can now prove that the macro and VIPER levels are identical, for this value of DSF.

The cases for DSF = (F, F, T) and (F, T, F) can be proven using the same tactic. For DSF = (F, T, T), (T, F, F), or (T, F, T), the proofs are simpler since for each case an error condition is generated, which causes execution to stop.

These error conditions are expressed with respect to the macro level by the following theorem:
\[
\begin{align*}
\text{F write_reg_illegalpdest_aux} &= \neg (\text{rep:'rep_ty}) (a:\\text{wordn}) (x:\\text{wordn}) (y:\\text{wordn}) (p:\\text{wordn}) (b:\\text{bool}) \\
& \quad (\text{stop:\text{bool}) (ir:\text{wordn}) (ram:\text{memory}) (value:\text{wordn}) (newb:\text{bool}) \\
& \quad (((\text{DSF rep ir}) = (F,T,T)) \lor \\
& \quad ((\text{DSF rep ir}) = (T,F,F)) \lor \\
& \quad ((\text{DSF rep ir}) = (T,F,T)))) \\
& \quad \implies \\
& \quad (\text{write_reg rep (a, x, y, p, b, stop, ir, ram, value, newb) = (a, x, y, p, b, T, ram}))
\end{align*}
\]

and specializing it for SHLB:

\[
\begin{align*}
\text{F illegal_shlb} &= (\text{SPECL ["rep:'rep_ty"; } \\
\text{"a:*wordn"; } \\
\text{"x:*wordn"; } \\
\text{"y:*wordn"; } \\
\text{"add (rep:'rep_ty) (p, wordn rep I)"; } \\
\text{"b:bool"; } \\
\text{"F"; } \\
\text{"fetch (rep:'rep_ty) (ram, address rep p)"; } \\
\text{"ram:*memory"; } \\
\text{"shlb (rep:'rep_ty) } \\
\text{((load_r rep)} \\
\text{ (a,x,y,add rep(p,wordn rep 1), } \\
\text{fetch rep(ram,address rep p))), b); "} \\
\text{b:bool"] } \\
\text{write_reg_illegalpdest_aux));}
\end{align*}
\]

In the VIPER level the error conditions corresponding to DSF = (F, T, T), (T, F, F), and (T, F, T) are expressed by the theorem in Figure 8.4-5. Hence the proofs of equivalence for the cases resulting in errors consist of rewriting the goals using the tactic shown in Figure 8.4-6.

We now have proven the goal that the macro level correctly implements the shift-left behavior at the VIPER instruction level:

\[
\begin{align*}
\text{F ! (rep:'rep_ty) a:*wordn (x:*wordn) (y:*wordn) p:*wordn b:bool) (stop:bool) (ram:*memory) \\
& \quad (\neg (\text{DSF rep (fetch rep (ram, (address rep p)))) \land \\
& \quad (\neg (\text{DSF rep (fetch rep (ram, (address rep p)))=(T,T,F)) \land \\
& \quad (\neg (\text{DSF rep (fetch rep (ram, address rep p))=(T,T,T)) \land \\
& \quad (\text{FSF rep (fetch rep (ram, address rep p)) = (T,T,F,F)) \land \\
& \quad (\text{MSF rep (fetch rep (ram, address rep p)) = (T,T)))}) = \\
& \quad (\text{SHLB rep (a, x, y, p, b, stop, ram) = } \\
& \quad \text{cohn_NEXT rep (a, x, y, p, b, stop, ram))})
\end{align*}
\]

64
Figure 8.4-5: Error cases in VIPER specification

```
let newp = (add rep (p, string rep I)) in
let io = ((cohn_OUTPUT rep (dsf, csf)) \n         (cohn_INPUT rep (dsf, csf, fsf))) in
let r = cohn_REG rep (rsf, a, x, y, newp) in
let m = cohn_MEMREAD rep (ram, msf, addr, x, y, io, cohn_NILM rep (dsf, csf, fsf)) in
let aluout = cohn_ALU rep (fsf, msf, dsf, r, m, b) in
let newp = (add rep (p, string rep I)) in
((\"stop\") \n (\"csf\") \n (valid_address rep newp) \n (\"(dsf = (T,T,T))\") \n (\"(dsf = (T,T,F))\") \n (dsf = (F,T,T)) \n (fsf = (T,T,F,F))) =>
 (cohn_NEXT rep (a, x, y, p, b, F, ram) =
  (a, x, y, newp, b, T, ram)))
```

Figure 8.4-6: Tactics used in proof of SHLB
8.5 DEFINITION OF THE DECODER

It was mentioned above that the mapping from the 12-bit opcode field of the VIPER level to the 20 orthogonal instructions of the macro level is effected by a decoder. We have specified the decoder in terms of 24 cases, corresponding to the 20 instructions in the macro level, 3 error cases, and an extra case for the NOOP instruction. To complete the verification of the macro level it is shown that the cases associated with the macro-level instructions are exactly the preconditions for these instructions. Also, it is shown that the cases cover all the possible values for the VIPER opcode field. The cases for the decoder are given in Appendix J.
9.0 CONCLUSIONS

This task was initiated because previous attempts to verify the design of the VIPER microprocessor using mechanical theorem provers were not completed. Since Cohn's incomplete verification effort was published in its entirety, we had the opportunity to attempt to determine why it was so difficult to complete. One reason is the large jump in abstraction between the instruction specification and the implementation. The second reason is the complexity of the specification itself. Many machines have clearly identified instructions with orthogonal fields to define addressing modes, register selection, etc. This is not the case for the VIPER architecture. Thus, although the instruction architecture is not complex, 122 unique cases must be separately considered in verifying the implementation. Each of the cases considered in Cohn's VIPER verification effort required approximately one person-week to complete.

Based on the success Windley achieved using a hierarchical methodology to verify a simpler microprocessor (AVM-1), we decided to apply the methodology to the VIPER processor. Windley's methodology depends on viewing the design of a microprocessor as a hierarchy of interpreters, the topmost providing the abstraction of the instructions accessible to the assembly language programmer and the lowest the implementation that is to be verified. A reasonable choice for the lowest level is an abstraction of the microprocessor that consists of its blocks such as the ALU, registers and latches; the original proof effort for the VIPER processor used this as the level to be verified and referred to it as the electronic block model. Among the choices for intermediate levels using the Windley methodology is an interpreter of microinstructions, which captures the decision that the microprocessor is microcoded.

The VIPER design is not microcoded, because the designers concluded that a hardwired design is faster than one where the control is achieved through microprograms. Moreover, the VIPER design does not suggest any convenient levels other than the instruction level and the electronic block model. Consequently, the Windley methodology could not be applied to the VIPER design. What our verification effort is concerned with is a microcoded design that we developed to realize the VIPER instruction set: the electronic block model of our design is approximately equivalent in complexity to that of RSRE's design.

To address the issue of the complexity of the specification, we introduced a level below the VIPER instruction level, which provides the same functionality but in terms of 20 orthogonal instructions. Of course VIPER programs will not run on this 20-instruction level, so it remained to
show the equivalence of this new level with the VIPER instruction-set level. Our design consisted of 5 levels and entailed the verification of the four lowest of these.

Our verification demonstrates the following: Corresponding to a VIPER object program instruction occupying the 12-bit opcode field, the logic of the electronic-block model is such that the correct ALU function will be invoked, the arguments (if any) will be drawn from the correct register and main memory locations, the results (if any) will be stored in the correct register (and flag bit) or main memory location, and the program counter will be correctly updated (incremented by one or set to the correct jump address). Since our design is microcoded, the proof entails (among many other things) showing that the microprogram corresponding to each instruction is correct.

What the verification does not guarantee is important to disclose:

- Our specification of the electronic block-model does not capture the semantics of the low-level functions, such as add, shift-left, xor, etc. These functions are not defined. Hence, it is not possible to use our specifications to reason about the computations of assembly language programs. We could have easily provided a semantics for these functions as a specification to be verified of an implementation more concrete than the electronic-block model. We decided not to provide such a specification, as our main goal was to verify the control logic of the microprocessor. This was also a criticism of Cohn's specification, but Cohn's came closer than ours in capturing the semantics of the operations.

- Emphasizing what was indicated above, we have not verified an implementation more concrete than the electronic-block model, such as a gate-level implementation. It is not clear that verification is the most cost-effective approach to checking gate-level descriptions.

- For simplicity we have assumed a single compare function. The VIPER processor has 16 compare instructions, but the logic to realize these differ in only trivial ways.

- The VIPER processor has external control lines; such as a reset button. The RSRE specification does not consider these lines, nor do we.

- Our specification (similar to RSRE's) does not deal with how long an instruction takes to execute. Handling timing specifications is feasible, but would severely complicate the verification. Again, other techniques are better suited to reasoning about timing for the relatively simple control logic that the VIPER processor employs.

- We have assumed that the main memory responds essentially instantaneously to read or write requests. VIPER can support an asynchronous interaction between the processing unit and
main memory. Techniques are known for modeling such an interaction, but we did not use them here.

- Main memory is assumed to be a black box. It is certainly feasible to consider a less abstract model of memory, such as one that models the decoder, sense lines, etc. Again, verification is not the best approach to reason about the details of a memory system.

Our major goal was to determine if the verification of such a hierarchical design is simpler than the verification of a flat design, such as VIPER. Furthermore, we wanted to determine if any gain is achieved through introduction of an orthogonal instruction set. The most difficult aspect of the Cohn verification effort was the consideration of the 122 cases that are part of the RSRE specification. Of course, our verification had to face these same 122 cases, but the objects being verified with respect to these cases is much "closer" to the specification than was the case for Cohn's proof. Having completed the verification we conclude that the methodology can simplify microprocessor verification efforts.

A second goal was to determine if, through the use of the hierarchical methodology and a previous successful verification effort of a simpler microprocessor, the verification of a larger microprocessor would be less of a tour de force than has been the experience with previous verification efforts. Towards this goal, the main contributors of the project team were two Master-level students, with skills in logic but no previous experience with formal methods or mechanical theorem provers. Moreover, the proof effort was divided up – each student assuming responsibility for two levels. Although each of the students completed his task, their work did not compose. Each student made assumptions about the the micro level, but in a few instances without communicating them to the other. In the end, these changes required most of the proof to be redone – and in the absence of those who carried out the initial proof. If the communication between the human provers had been better, much grief would have been avoided.

A third goal was to determine if, through the use of special-purpose HOL tactics, the proof could have been accomplished with less human intervention. (HOL is mostly a proof checker, as compared with the Boyer Moore theorem prover. Excessive human intervention is avoided through the employment of tactics that match the expressions being reasoned about.) Towards this goal, we developed a few symbolic execution tactics intended to cover the actions associated with the implementation of an instruction, e.g., a microinstruction, phase instruction, or macro level instruction. At the lowest levels, special-purpose symbolic execution tactics worked perfectly, in effect handling all cases. At the upper levels we were less successful, requiring hand-crafted tactics
corresponding to each instruction class (shift, write memory, arithmetic, etc). At the highest level (proof of macro to RSRE specification), we were able to reuse very few tactics, a statement about the irregularity of the VIPER instruction set.

A final goal was to determine the effectiveness of HOL for a large proof effort. The proof was completed, but it was painful. The experience of HOL users has been that human proof time vastly exceeds HOL's processing time. This was not our experience with this proof. We generated expressions that sometimes consumed hours of processing time to reason about.

Additional issues to be studied include:

- The scalability of the proof effort. Our team would not have been willing to tackle a microprocessor an order of magnitude more complex than the VIPER architecture. The discovery of tactics that handle most cases would, of course, simplify the human effort.

- Reasoning about changes. Most of the HOL processing time and a large fraction of the human time was devoted to re-doing proofs subsequent to design changes. Identifying those parts of a proof that need not be redone would have saved vast effort.

- The role of a simulator to discover "obvious" errors. We designed the microprocessor, but never tested it. Hence, the verification effort detected errors that would have been discovered with the most rudimentary of tests. Not having access to a CAD system with a design simulator for HOL specifications, we should have written a simulator in ML.

- The role of correctness-preserving transformations to transform a verified micro-coded design into a more efficient hardwired design.

Further work is needed before it can be concluded that larger microprocessors can be verified and that the hierarchical interpreter theory offers benefits in such efforts. Work underway at Boeing on the verification of a fault-tolerant processor gives promise of another data point. Clearly, the interpreter theory organizes the proof, but still the number of cases that the verifier must consider is staggering. There are too many cases to be handled individually. Hand-crafted tactics can be constructed to allow the HOL system to process many cases in one shot, but we discovered that the performance of the theorem prover was dismal. The use of special Boolean decision packages should be of considerable help.
REFERENCES


HOL is a general theorem-proving system developed at the University of Cambridge (ref. 19, 20) that is based on Church's theory of simple types, or higher-order logic (ref. 21). Church developed higher-order logic as a foundation for mathematics, but it can be used for describing and reasoning about computational systems of all kinds. Higher-order logic is similar to the more familiar predicate logic, but allows quantification over predicates and functions, not just variables, allowing more general systems to be described.

HOL grew out of Robin Milner's LCF theorem prover (ref. 22) and is similar to other LCF progeny such as NUPRL (ref. 23). Because HOL is the theorem-proving environment used in the body of this work, we will describe it in more detail.

HOL's proof style can be tailored to the individual user, but most users find it convenient to work in a goal-directed fashion. HOL is a tactic-based theorem prover. A tactic breaks a goal into one or more subgoals and provides a justification for the goal reduction in the form of an inference rule. Tactics perform tasks such as induction, rewriting, and case analysis. At the same time, HOL allows forward inference and many proofs are a combination of both forward and backward proof styles. Any theorem-proving strategy a user employs in connection with HOL is checked for soundness, eliminating the possibility of incorrect proofs.

HOL provides the user with a metalanguage, ML, for programming and extending the theorem prover. Using ML, tactics can be put together to form more powerful tactics, new tactics can be written, and theorems can be combined into new theories for later use. The metalanguage makes the HOL verification system extremely flexible.

In HOL, all proofs, even tactic-based proofs, are eventually reduced to the application of inference rules. Most nontrivial proofs require large numbers of inferences. Proofs of large devices such as microprocessors can take many millions of inference steps. In a proof containing millions of steps, what kind of confidence do we have that the proof is correct? One of the most important features of HOL is that it is secure, meaning that new theorems can only be created in a controlled manner. HOL is based on five primitive axioms and eight primitive inference rules. All high-level inference rules and tactics do their work through some combination of the primitive inference rules. Because the entire proof can be reduced to one using only eight primitive inference rules and five primitive axioms, an independent proof-checking program could check the proof syntactically.
The Language.

The object language of HOL is described in this section. We will discuss HOL's terms and types.

Terms. All HOL expressions are made up of terms. There are four kinds of terms in HOL: variables, constants, function applications, and abstractions (lambda expressions). Variables and constants are denoted by any sequence of letters, digits, underlines, and primes starting with a letter. Constants are distinguished in the logic; any identifier that is not a distinguished constant is taken to be a variable. Constants and variables can have any finite arity, not just 0, and, thus, can represent functions as well.

Function application is denoted by juxtaposition, resulting in a prefix syntax. Thus, a term of the form "t1 t2" is an application of the operator t1 to the operand t2. The term's value is the result of applying t1 to t2.

An abstraction denotes a function and has the form "A x. t". An abstraction "A x. t" has two parts: the bound variable x and the body of the abstraction t. It represents a function, f, such that "f(x) = t". For example, "A y. 2*y" denotes a function on numbers which doubles its argument.

Constants can belong to two special syntactic classes. Constants of arity 2 can be declared to be infix. Infix operators are written "rand1 op rand2" instead of in the usual prefix form: "op rand1 rand2". Table A-1 shows several of HOL's built-in infix operators.

Constants can also belong to another special class called binders. A familiar example of a binder is "\(\). If c is a binder, then the term "c x.t" (where x is a variable) is written as shorthand for the term "c(\(x \). t)". Table A-2 shows several of HOL's built-in binders.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Application</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>t1 = t2</td>
<td>t1 equals t2</td>
</tr>
<tr>
<td></td>
<td>t1,t2</td>
<td>the pair t1 and t2</td>
</tr>
<tr>
<td>∧</td>
<td>t1 ∧ t2</td>
<td>t1 and t2</td>
</tr>
<tr>
<td>∨</td>
<td>t1 ∨ t2</td>
<td>t1 or t2</td>
</tr>
<tr>
<td>⇒</td>
<td>t1 ⇒ t2</td>
<td>t1 implies t2</td>
</tr>
</tbody>
</table>
In addition to the infix constants and binders, HOL has a conditional statement that is written $a \rightarrow b \mid c$, meaning "if $a$, then $b$, else $c$.”

**Types.** HOL is strongly typed to avoid Russell’s paradox and others like it. Russell’s paradox occurs in a high order logic when one can define a predicate that leads to a contradiction. Specifically, suppose that we define $P$ as $P(x) = \neg x(x)$ where $\neg$ denotes negation. $P$ is true when its argument applied to itself is false. Applying $P$ to itself leads to a contradiction since $P(P) = \neg P(P)$ (i.e., $true = false$). This kind of paradox can be prevented by typing since, in a typed system, the type of $P$ would never allow it to be applied to itself.

Every term in HOL is typed according to the following recursive rules:

- a. Each constant or variable has a fixed type.
- b. If $x$ has type $\alpha$ and $t$ has type $\beta$, the abstraction $\lambda x. t$ has the type $(\alpha \rightarrow \beta)$.
- c. If $t$ has the type $(\alpha \rightarrow \beta)$ and $u$ has the type $\alpha$, the application $t \ u$ has the type $\beta$.

Types in HOL are built from type variables and type operators. Type variables are denoted by a sequence of asterisks (*) followed by a (possibly empty) sequence of letters and digits. Thus, *, **, and *ab2 are all valid type variables. All type variables are universally quantified implicitly, yielding type polymorphic expressions.

Type operators construct new types from existing types. Each type operator has a name (denoted by a sequence of letters and digits beginning with a letter) and an arity. If $\sigma_1, \ldots, \sigma_n$ are types and $\text{op}$ is a type operator of arity $\nu$, the $(\sigma_1, \ldots, \sigma_n)\text{op}$ is a type. Note that type operators are postfix while normal function application is prefix or infix. A type operator of arity 0 is a type constant.

HOL has several built-in types which are listed in table A-3. The type operators bool, ind, and fun are primitive. HOL has a special syntax that allows $(*, **)\text{prod}$ to be written as $(\star \ # \ **), (\ast, **)\text{sum}$ to be written as $(\ast \ + \ **)$, and $(\ast, **)\text{fun}$ to be written as $(\ast \rightarrow **).
Table A-3: HOL Type Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Arity</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>0</td>
<td>booleans</td>
</tr>
<tr>
<td>ind</td>
<td>0</td>
<td>individuals</td>
</tr>
<tr>
<td>num</td>
<td>0</td>
<td>natural numbers</td>
</tr>
<tr>
<td>(*)list</td>
<td>1</td>
<td>lists of type *</td>
</tr>
<tr>
<td>(*,**)prod</td>
<td>2</td>
<td>products of * and **</td>
</tr>
<tr>
<td>(*,**)sum</td>
<td>2</td>
<td>coproducts of * and **</td>
</tr>
<tr>
<td>(*,**)fun</td>
<td>2</td>
<td>functions from * to **</td>
</tr>
</tbody>
</table>

The Proof System.

HOL is not an automated theorem prover but is more than simply a proof checker, falling somewhere between these two extremes. HOL has several features that contribute to its use as a verification environment:

a. Several built-in theories, including booleans, individuals, numbers, products, sums, lists, and trees. These theories contain the five axioms that form the basis of higher order logic as well as a large number of theorems that follow from them.

b. Rules of inference for higher order logic. These rules contain not only the eight basic rules of inference from higher order logic, but also a large body of derived inference rules that allow proofs to proceed using larger steps. The HOL system has rules that implement the standard introduction and elimination rules for Predicate Calculus as well as specialized rules for rewriting terms.

c. A collection of tactics. Examples of tactics include: REWRITE_TAC which rewrites a goal according to some previously proven theorem or definition; GEN_TAC which removes unnecessary universally quantified variables from the front of terms; and EQ_TAC which says that to show two things are equivalent, we should show that they imply each other.

d. A proof management system that keeps track of the state of an interactive proof session.

e. A metalanguage, ML, for programming and extending the theorem prover. Using the metalanguage, tactics can be put together to form more powerful tactics, new tactics can be written, and theorems can be aggregated to form new theories for later use. The metalanguage makes the verification system extremely flexible.
Appendix B: INTERPRETER THEORY AND ABSTRACT FUNCTIONS

File: def_aux.ml

Description: Defines generic functions used in subsequent Viper specifications.

File: def_aux.ml

set_search_path (search_path() @ lib_dir_list);

loadf 'abstract'; new_theory 'aux_def';
new_parent 'tuple';
new_type_abbrev('time',":num");

let abs_rep = new_abstract_representation [
  % ALU functions %
  % negation %
  ('neg', ":(#wordn -> *wordn) *)
  ;
  % addition without carry %
  ('add', ":(#wordn # *wordn -> *wordn) *
  ;
  % predicate carry for addc %
  ('addp', ":(#wordn # *wordn # *wordn) -> bool ")
  ;
  % overflow predicate for add %
  ('aovfl', ":(#wordn # *wordn # *wordn) -> bool ")
  ;
  % subtract %
  ('sub', ":(#wordn # *wordn) -> *wordn ")
  ;
  % carry predicate for sub %
  ('subp', ":(#wordn # *wordn # *wordn) -> bool ")
  ;
  % overflow predicate for sub %
  ('sovfl', ":(#wordn # *wordn # *wordn) -> bool ")
  ;
  % bitwise xor %
  ('bxor', ":(#wordn # *wordn -> *wordn) ")
  ;
  % bitwise and %
  ('band', ":(#wordn # *wordn -> *wordn) ")
  ;
% bitwise nor %
('bnor', ":(wordn # wordn -> wordn)"
)
%
% bitwise not %
('bnot', ":(wordn -> wordn)"
)
%
% bitwise or %
('bor', ":(bool # bool -> bool)"
)
%
% SHIFTER functions %
% shift right , copy sign bit %
('shr', ":(wordn -> wordn)"
)
%
% shift left %
('shl', ":(wordn -> wordn)"
)
%
% shift right thru b %
('shrb', ":(wordn # bool -> wordn)"
)
%
% shift left thru b %
('shlb', ":(wordn # bool -> wordn)"
)
%
% Coercion functions %
% numeric vaule of n-bit word %
('val', ":(wordn -> num)"
)
%
% wordn representation of number %
('wordn', ":(num -> wordn)"
)
%
% address part of a word %
('address', ":(wordn -> address)"
)
%
% address converting to a word %
('pad', ":(address -> wordn)"
)
%
% combine msb opcode bits and lsb address bits to wordn %
('join', ":((opcode # address) -> wordn)"
)
%
% Test functions %
% see if address is valid %
('valid_address', ":(wordn -> bool)"
)
%
% decoder %
('decode', ":((opcode # bool) -> (bool # bit5 # bool))"
)
%
% Compare function %
% cmp two words depending on code %
('bcmp', ":(*wordn # wordn # bool # bt4 -> bool)"
)
;

% Subranging functions %
% opcode portion of word %
('opcode', ":(*wordn -> *opcode)"
)
;
% retrieve bit0 of a *wordn %
('bit0', ":(*wordn -> bool)"
)
;
% retrieve bitn of a *wordn %
('bitn', ":(*wordn -> bool)"
)
;
% retrieve rsf of a *wordn %
('RSF', ":(*wordn -> bt2)"
)
;
% retrieve msf of a *wordn %
('MSF', ":(*wordn -> bt2)"
)
;
% retrieve dsf of a *wordn %
('DSF', ":(*wordn -> bt3)"
)
;
% retrieve csf of a *wordn %
('CSF', ":(*wordn -> bool)"
)
;
% retrieve fsf of a *wordn %
('FSF', ":(*wordn -> bt4)"
)
;

% Memory functions %
% fetch a word from memory %
('fetch', ":((*memory # *address) -> *wordn)"
)
;
% store a word in memory %
('store', ":((*memory # *address # *wordn) -> *memory)"
)
;
% fetch a word from io % memory mapped io %
('fetchio', ":((*memory # *address) -> *wordn)"
)
;
% store a word in memory % memory mapped io %
('storeio', ":((*memory # *address # *wordn) -> *memory)"
)
};

close_theory();

79
%-- generic_interpreter.ml

Author: (c) P. J. Windley 1990

Date: 09 JAN 90

Description:

Defines a generic interpreter used in subsequent specifications.
The interpreter is proven to be correct under certain obligations.
The interpreter in this file is synchronous.

set_search_path (search_path() @ lib_dir_list);

system '/bin/rm gen_I.th';

new_theory 'gen_I';

map loadf ['abstract'];

new_type_abbrev('time',':num');

new_type_abbrev('time',':num');

Generic specification

let cpu_abs = new_abstract_representation

  [ ('inst_list',':(key*(state->env->state))list')
    ; ('key',':key->num')
    ; ('select',':state->env->key')
    ; ('cycles',':key->num')
    ; ('substate',':state->state')
    ; ('subenv',':env->env')
    ; (' Impl',':(time->state)->(time->env)->bool')
    ; ('count',':state->env->key')
    ];
('start',":*key")
;
];

make_inst_thms cpu_abs;;

let I_rep_ty = abstract_type 'gen_I' 'key';;

let INTERP_def = new_definition
('INTERP',
"! (rep:'I_rep_ty) (s:time->*state) (e:time->*env) .
INTERP rep s e =
  !t:time.
  let n = (key rep (select rep (s t) (e t))) in 
  s(t+1) = (SND (EL n (inst_list rep))) (s t) (e t)"
);;

let INTERP_DEF_EXPANDED = EXPAND_LET_RULE INTERP_def;;

%<
let FIND = new_recursive_definition
  false
  list_Axiom
  'FIND'
  "(FIND x [] = 0) /
  (FIND x (CONS h t) =
     (x = h) => 0 | 1 + (FIND x t))"
);;

letrec pos x 1 =
  null 1 => 0 |
  (x = (hd 1)) => 1 | (1 + (pos x (tl 1)));;
>
%

let impl_imp_def = new_definition
('IMPL_IMP',
"! inst:*key#(*state->*env->*state))
(s':time'->*state')
(e':time'->*env') .
IMPL_IMP rep s' e' inst =
(Impl (rep:'I_rep_ty) s' e') =>
  (!t:time'.
    let s = (\t. (substate rep (s' t))) in
    let e = (\t. (subenv rep (e' t))) in
    let c = (cycles rep (select rep (s t) (e t))) in 
    (select rep (s t) (e t) = (FST inst)) /
    (count rep (s' t) (e' t) = (start rep)) =>
    ((SND inst) (s t) (e t) = (s (t + c))) /
    (count rep (s' (t + c)) (e' (t + c)) = (start rep)))"
let IMPL_IMP_EXPANDED = EXPAND_LET_RULE impl_imp_def;;

new_theory_obligations
[
  "EVERY (IMPL_IMP (rep:'I_rep_ty) (s':time'->state') (e':time'->env'))
   (inst_list rep)"
;
  "!k:*key. (key (rep:'I_rep_ty) k) < (LENGTH (inst_list rep))"
;
  "!k:*key . k = (FST (EL (key (rep:'I_rep_ty) k) (inst_list rep)))"
];;

let IMPL_NEXTSTATE_LEMMA = TAC_PROOF
(([]).
  "let s = (\t:time .(substate rep (s' t))) and
  e = (\t:time .(subenv rep (e' t))) in (.
  (Impl (rep:'I_rep_ty)) s' e' =>
  (\t:time'.
  (count rep (s' t) (e' t) = (start rep)) =>
  ((substate rep (s' (t+(cycles rep (select rep (s t) (e t)))))) =
  (SND (EL (key rep (select rep (s t) (e t)))
   (inst_list rep))) (s t) (e t))))",
EXPAND_LET_TAC
THEN REPEAT STRIP_TAC
THEN POP_ASSUM_LIST (\asl .
  let asl' =
   map (PURE_REWRITE_RULE [EVERY_EL;IMPL_IMP_EXPANDED]) asl in
  MAP_EVERY ASSUME_TAC
  (map
   (\thm.
    (SPEC "(key (rep:'I_rep_ty)
      (select rep
      (substate rep(s' t))
      (subenv rep (e' t))))" thm) ?
    (SPEC "(select (rep:'I_rep_ty)
      (substate rep(s' t))
      (subenv rep (e' t)))" thm) ?
    thm) asl'))
  THEN RES_TAC
  THEN POP_ASSUM (\thm. ASSUME_TAC (REWRITE RULE [] (SPEC "t:time''" thm)))
  THEN RES_TAC
  THEN FIRST_ASSUM (ACCEPT_TAC o SYM_RULE)
let IMPL_NEXTSTATE_LEMMA_EXPANDED = EXPAND_LET_RULE IMPL_NEXTSTATE_LEMMA;;

let time_shift = new_prim_rec_definition
  ('time_shift',
   "(time_shift f (s:time->*state) (e:time->*env) 0 = 0) /
   (time_shift f s e (SUC n) = (let t = (time_shift f s e n) in
    t + (f (s t) (e t)))")
  );;

let I_CLOCK_LEMMA = TAC_PROOF
  (([],
    "let s = (\t:time .(substate rep (s' t))) and
    e = (\t:time. (subenv rep (e' t))) in (Impl rep) s _ e'
    ((count rep) (s' 0) (e' O) = (start rep)) ==>
    !t. let t_impl =
      (time_shift (\st env. (cycles rep (select rep st env))) s e t) in
    (count (rep:"I_rep_ty)) (s' t_impl) (e' t_impl) = (start rep))")
  EXPAND_LET_TAC
  THEN REPEAT GEN_TAC
  THEN STRIP_TAC
  THEN INDUCT_TAC
  THEN REWRITE_TAC (time shift: o_DEF;LET_DEF)
  THEN (FIRST_ASSUM ACCEPT_TAC ORELSE ALL_TAC)
  THEN POP_ASSUM (
thm. ASSUME_TAC
    (CONV_RULE (TOP DEPTH_CONV BETA_CONV)
      (ONCE_REWRITE_RULE [o_DEF] thm)))
  THEN BETA_TAC
  THEN POP_ASSUM_LIST (\asl .
    let asl' =
      map (PURE_REWRITE_RULE [EVERY_EL;IMPL_IMP_EXPANDED]) asl in
      MAP_EVERY ASSUME_TAC
      (map
        (\thm.
          (SPEC "(key (rep:"I_rep_ty)
            (select rep
              (substate rep
                (s'
                  (time_shift
                    (\st env. cycles rep(select rep st env))
                    (\'t'. substate rep(s' t')))
                    (\t'. subenv rep (e' t')) t)))
                  (subenv rep
                    (s' t')))))
      asl')
  ...
((e' \\
  (time_shift \\
    (\st env. cycles rep(select rep st env)) \\
      (\t'. substate rep(s' t'))) \\
    (\t'. subenv rep (e' t'))) t))))" thm ) ? \\
    (SPEC "((select (rep:"I_rep_ty) \\
                          (substate rep \\
                          (s' \\
                            (time_shift \\
                              (\st env. cycles rep(select rep st env)) \\
                                (\t'. substate rep(s' t'))) \\
                              (\t'. subenv rep (e' t'))) t))))" thm ) ? \\
    (SPEC "((select (rep:"I_rep_ty) \\
                          (substate rep \\
                          (e' \\
                            (time_shift \\
                              (\st env. cycles rep(select rep st env)) \\
                                (\t'. substate rep(s' t'))) \\
                              (\t'. subenv rep (e' t'))) t))))" thm ) ? \\
    (SPEC "((select (rep:"I_rep_ty) \\
                          (substate rep \\
                          (e' \\
                            (time_shift \\
                              (\st env. cycles rep(select rep st env)) \\
                                (\t'. substate rep(s' t'))) \\
                              (\t'. subenv rep (e' t'))) t))))" thm ) ? \\
THEN RES_TAC \\
THEN POP_ASSUM (\thm. ASSUME_TAC (REWRITE_RULE []) \\
    (SPEC "((time_shift \\
                  (\st env. cycles (rep:"I_rep_ty) (select rep st env)) \\
                  (\t'. substate rep(s' t'))) \\
                  (\t'. subenv rep (e' t'))) t):time" thm )) \\
THEN RES_TAC \\
);; \\
let I_CLOCK_LEMMA_EXPANDED = EXPAND_LET_RULE I_CLOCK_LEMMA;; \\
let IMPL_I_CORRECT = prove_thm ("IMPL_I_CORRECT", \\
  "let s = (\t:t:time .(substate rep (s' t))) and \\
    e = (\t:t:time .(subenv rep (e' t))) in \\
    (Impl rep) s' e' /
    ((count (rep:"I_rep_ty)) (s' 0) (e' 0) = (start rep)) => \\
    let f = time_shift (\st env. (cycles rep (select rep st env))) s e in \\
    (INTERP rep) (s o f) (e o f)", EXPAND_LET_TAC \\
    THEN REPEAT GEN_TAC \\
    THEN PURE_REWRITE_TAC [INTERP_DEF_EXPANDED;o_DEF] \\
    THEN STRIP_TAC \\
    THEN IMP_RES_TAC (PURE_ONCE_REWRITE_RULE [o_DEF] I_CLOCK_LEMMA_EXPANDED) \\
    THEN GEN_TAC \\
    THEN BETA_TAC \\

84
THEN PURE_ONCE_REWRITE_TAC
  [EXPAND_LET_RULE (REWRITE_RULE [ADD1] time_shift)]
THEN BETA_TAC
THEN POP_ASSUM (\x. ASSUME_TAC (SPEC "t:time" x))
THEN IMP_RES_TAC IMPL_NEXTSTATE_LEMMA_EXPANDED
);

close_theory();;
File: mk_aux.ml

Description: Prove auxiliary theorems used in subsequent proofs.

system '/bin/rm aux_thms.th';
new_theory 'aux_thms';

Auxiliary list definitions and theorems

let SET_EL_DEF = new_prim_rec_definition
('SET_EL_DEF',
 "(SET_EL 0 (lst:(*)) x = (CONS x (TL lst))) /
 (SET_EL (SUC n) lst x = (CONS (HD lst) (SET_EL n (TL lst) x)))"
);

let SET_EL = prove_thm
('SET_EL',
 "! h t x .
 (SET_EL 0 (CONS h t) x = (CONS x t)) /
 (SET_EL (SUC n) (CONS h t) x = (CONS h (SET_EL n t x)))",
REPEAT GEN_TAC
THEN REWRITE_TAC [SET_EL_DEF; HD; TL]
);

let EL_SET_EL = prove_thm
('EL_SET_EL',
 "! x n lst . EL n (SET_EL n lst x) = x",
GEN_TAC
THEN INDUCT_TAC
THEN REWRITE_TAC [SET_EL_DEF; EL; CONS; TL; HD]
THEN LIST_INDUCT_TAC
THENL [ POP_ASSUM (\x. ASSUME_TAC (SPEC "TL[:(,)list] x")) ; ALL_TAC ]
THEN ASM_REWRITE_TAC [TL]
);

close_theory();
File: threval.ml

Description: Defines a new type 'triple' with members ONE, TWO, THREEx used to instantiate *key in the EBM to Phase level proof of viper.

%------------------------------------------------------------------------

system '/bin/rm -f threval.th';;
new_theory 'threeval';;
let triple = define_type 'triple' 'triple = ONE | TWO | THREE';;
let y = prove_constructors_distinct triple;;
let triple_induct = prove_induction_thm triple;;
let triple_cases = prove_cases_thm triple_induct;;
let triple_value = new_definition('triple_value','
"!x:triple. triple_value x = (x=ONE) => 0 |
   (x=TWO) => 1 |
   2"
');;
let triple_VALUE_LEMMA = prove_thm
  ('triple_VALUE_LEMMA',
"(triple_value ONE = 0) /\ (triple_value TWO = 1)
 /\ (triple_value THREE = 2)",
REWRITE_TAC[triple_value] THEN
STRIP_ASSUME_TAC y THEN
ASSUM_LIST(\asl. REWRITE_TAC[NOT_EQ_SYM (el 1 asl); NOT_EQ_SYM (el 2 asl);
   NOT_EQ_SYM (el 3 asl)])
);

let triple_LENGTH_LEMMA = prove_thm
  ('triple_LENGTH_LEMMA',
"! x:triple (l1 l2 l3:=). triple_value x < (LENGTH [l1; l2; l3])",
REPEAT GEM_TAC THEN REWRITE_TAC [LENGTH] THEN
REWRITE_TAC[triple_value] THEN
COND_CASES_TAC THENL [
   REWRITE_TAC[LESS_0] ;
   COND_CASES_TAC THENL [
      CONV_TAC(TOP_DEPTH_CONV num_CONV )
      THEN REWRITE_TAC[LESS_MONO_EQ;LESS_0] ;
   ]
)
CONV_TAC(TOP_DEPTH_CONV num_CONV)
  THEN REWRITE_TAC[LESS_MONO_EQ;LESS_0]

);;

close_theory();;

Appendix C: VIPER LEVEL SPECIFICATION

% Prove that the macro level ==> cohn level
% system '/bin/rm cohn_eqvaux.th';;
set_search_path (search_path() @ lib_dir_list);
loadf 'abstract';;
new_theory 'cohn_eqvaux';;
new_parent 'aux_def';;
new_parent 'cohn_viper';;
new_parent 'macro_def';;
let rep_ty = abstract_type 'aux_def' 'opcode';;
let cohn_REG = definition 'cohn_viper' 'cohn_REG';;
let cohn_INVALID = definition 'cohn_viper' 'cohn_INVALID';;
let write_reg = EXPAND_LET_RULE (definition 'macro_def' 'write_reg');;
let load_r = EXPAND_LET_RULE (definition 'macro_def' 'load_r');;
let cohn_NEXT = definition 'cohn_viper' 'cohn_NEXT';;
let cohn_NEXT_expanded = EXPAND_LET_RULE cohn_NEXT;;

% register loads are eqv%
let reg_eqv = prove_thm
('reg_eqv',
"! (rep:'rep_ty) (a:*wordn) (x:*wordn) (y:*wordn) (p:*wordn)
  (b:bool) (ram:*memory) .
  (cohn_REG rep (RSF rep(fetch rep(ram,address rep p)),a,x,y,
    add rep(p,wordn rep 1))) =
  (load_r rep (a, x, y, add rep (p, wordn rep 1),
    fetch rep (ram, address rep p)))",
REPEAT GEN_TAC
THEN PURE_REWRITE_TAC [cohn_REG; load_r]
THEN REPEAT (COND_CASES_TAC THEN ASM_REWRITE_TAC [PAIR_EQ]);;

% cohn_stop %
let cohn_stop = prove_thm
('cohn_stop',
"! (rep:'rep_ty) (a:*wordn) (x:*wordn) (y:*wordn) (p:*wordn)
  (b:bool) (stop:bool) (ram:*memory) .

C-2
stop ==> (cohn_NEXT rep (a, x, y, p, b, stop, ram) = (a, x, y, p, b, T, ram))", REPEAT GEN_TAC THEN STRIP_TAC THEN ASM_REWRITE_TAC[cohn_NEXT_expanded];;

% cohn_noinc % let cohn_noinc = prove_thm ('cohn_noinc', "'(rep:rep_ty) (a:*wordn) (x:*wordn) (y:*wordn) (p:*wordn) (b:bool) (stop:bool) (ram:*memory) .
(let newp = (add rep (p, (ordn rep 1))) in
(((valid_address rep newp) \/
("stop) ==> (cohn_NEXT rep (a, x, y, p, b, stop, ram) = (a, x, y, newp, b, T, ram)))”,
REPEAT GEN_TAC THEN EXPAND_LET_TAC THEN STRIP_TAC THEN ASM_REWRITE_TAC [cohn_NEXT_expanded; cohn_INVALID]);;

% write_reg_illegalpdest_aux % let write_reg_illegalpdest_aux = prove_thm ('write_reg_illegalpdest_aux', "'(rep:rep_ty) (a:*wordn) (x:*wordn) (y:*wordn) (p:*wordn) (b:bool) (stop:bool) (ir:*aordn) (ram:*memory)
(value:*wordn (newb:bool).
((DSF rep ir) = (F,T,T)) \/
((DSF rep Jr) = (T,F,F)) \/
((DSF rep Jr) = (T,F,T))
==> (write_reg rep (a, x, y, p, b, stop, ir, ram, value, newb) = (a, x, y, p, b, T, ram))", REPEAT GEN_TAC THEN STRIP_TAC THEN ASM_REWRITE_TAC[write_reg; PAIR_EQ]);;

let THREE_TUPLE_VALUE_LEMMA = theorem 'tuple' 'THREE_TUPLE_VALUE_LEMA';;
let three_tuple_value_lemma = (SPECL ['"b:bt3"] THREE_TUPLE_VALUE_LEMA);
let bt3_remaining_lemma = prove_thm ('bt3_remaining_lemma', "'(bt3_remaining_lemma', ""! (b:bt3) .

90
\((\neg (b = (F,F,F))) \land \\
(\neg (b = (F,F,T))) \land \\
(\neg (b = (F,T,F))) \land \\
(\neg (b = (F,T,T))) \land \\
(\neg (b = (T,F,F))) \land \\
(\neg (b = (T,T,F))) \land \\
\Rightarrow (b = (T,F,T))\)

REPEAT GEN_TAC
THEN STRIP_TAC
THEN ASSUM_LIST \asl. ASSUME_TAC (REWRITE_RULE [(el I asl); (el 2 asl); (el 3 asl); (el 4 asl); (el 5 asl); (el 6 asl); (el 7 asl)];

\begin{align*}
\text{three_tuple_value_lemma})
\end{align*}
THEN ASM_REWRITE_TAC[];

let TWO_TUPLE_VALUE_LEMMA = theorem 'tuple ' TWO_TUPLE_VALUE_LEMMA;;

let two_tuple_value_lemma = (SPECL ['b:bt2'] TWO_TUPLE_VALUE_LEMMA);;

let bt2_remaining_lemma = prove_thm ('bt2_remaining_lemma',

"! (b:bt2) . \\
(\neg (b = (F,F))) \land \\
(\neg (b = (F,T))) \land \\
\Rightarrow (b = (T,T))",

REPEAT GEN_TAC
THEN STRIP_TAC
THEN ASSUM_LIST \asl. ASSUME_TAC (REWRITE_RULE [(el I asl); (el 2 asl); (el 3 asl)];

\begin{align*}
\text{two_tuple_value_lemma})
\end{align*}
THEN ASM_REWRITE_TAC[];

%
Author: Tony Leung

University of California, Davis

Prove that the macro level \Rightarrow cohn level
%
system '/bin/rm cohn_TTFF_aux.th';;

set_search_path (search_path() @ lib_dir_list);;
loadf 'abstract';;

new_theory 'cohn_TTFF_aux';;

new_parent 'aux_def';
new_parent 'cohn_viper';
new_parent 'macro_def';;

let rep_ty = abstract_type 'aux_def' 'opcode';;

let
cohn_ALU = EXPAND_LET_RULE (definition 'cohn_viper' 'cohn_ALU');;
let
cohn_SVAL = definition 'cohn_viper' 'cohn_SVAL';;
let
cohn_BVAL = definition 'cohn_viper' 'cohn_BVAL';;
let
cohn_VALUE = definition 'cohn_viper' 'cohn_VALUE';;
let
cohn_INVALID = definition 'cohn_viper' 'cohn_INVALID';;
let
cohn_ILLEGALCALL = definition 'cohn_viper' 'cohn_ILLEGALCALL';;
let
cohn_SPAREFUNC = definition 'cohn_viper' 'cohn_SPAREFUNC';;
let
cohn_ILLEGALPDEST = definition 'cohn_viper' 'cohn_ILLEGALPDEST';;
let
cohn_WRITE = definition 'cohn_viper' 'cohn_WRITE';;
let
cohn_ILLEGALWRITE = definition 'cohn_viper' 'cohn_ILLEGALWRITE';;
let
cohn_NILM = definition 'cohn_viper' 'cohn_NILM';;
let
cohn_NOOP = definition 'cohn_viper' 'cohn_NOOP';;
let
cohn_REG = definition 'cohn_viper' 'cohn_REG';;

let write_reg = EXPAND_LET_RULE (definition 'macro_def' 'write_reg');;

let load_r = EXPAND_LET_RULE (definition 'macro_def' 'load_r');;

let
cohn_NEXT = definition 'cohn_viper' 'cohn_NEXT';;
let
cohn_NEXT_expanded = EXPAND_LET_RULE cohn_NEXT;;

let write_reg = EXPAND_LET_RULE (definition 'macro_def' 'write_reg');;

% cohn_WRITE_TTFF %
let cohn_WRITE_TTFF = prove_thm
('cohn_WRITE_TTFF',
"%! (rep:'rep_ty) (a:*wordn) (x:*wordn) (y:*wordn) (p:*wordn)
(b:bool) (ram:*memory).
""(((DSF rep (fetch rep (ram, address rep p))) = (T,T,T)) \/
((DSF rep (fetch rep (ram, address rep p))) = (T,T,F))) ==> 
(cohn_WRITE rep (DSF rep (fetch rep (ram, address rep p)),
(CSF rep (fetch rep (ram, address rep p))))
 = F)",
REPEAT GEN_TAC
THEN STRIP_TAC
THEN ASM_REWRITE_TAC [cohn_WRITE; PAIR_EQ];;
let cohn_illegalcall_TTFF = prove_thm
('cohn_illegalcall_TTFF',
(((FSF rep (fetch rep (ram, address rep p))) = (T,T,F,F))
==>
((cohn_ILLEGALCALL rep
((DSF rep (fetch rep (ram, address rep p)));
(CSF rep (fetch rep (ram, address rep p))) /\ (FSF rep (fetch rep (ram, address rep p))) = F)))".
REPEAT GEN_TAC
THEN STRIP_TAC
THEN ASM_REWRITE_TAC[cohn_ILLEGALCALL; PAIR_EQ]);;

let cohn_NILM_TTFF = prove_thm
('cohn_NILM_TTFF',
"! (rep:'rep_ty) (a:*wordn) (x:*wordn) (y:*wordn) (p:*wordn) (b:bool) (ram:*memory).
((!(CSF rep (fetch rep (ram, address rep p))) /\
((DSF rep (fetch rep (ram, address rep p)) = T,T,F) /\
((DSF rep (fetch rep (ram, address rep p)) = T,F,F) /\
(FSF rep (fetch rep (ram, address rep p))) = (T,T,F,F))
==>
(cohn_NILM rep ((DSF rep (fetch rep (ram, address rep p)));
(CSF rep (fetch rep (ram, address rep p)));
(FSF rep (fetch rep (ram, address rep p))) = T))",
REPEAT GEN_TAC
THEN STRIP_TAC
THEN ASM_REWRITE_TAC[cohn_NILM; PAIR_EQ]);;

let cohn_sparefunc_TTFF = prove_thm
('cohn_sparefunc_TTFF',
"! (rep:'rep_ty) (a:*wordn) (x:*wordn) (y:*wordn) (p:*wordn) (b:bool).
(((FSF rep (fetch rep (ram, address rep p))) = (T,T,F,F)) ==>
(cohn_SPAREFUNC rep
((DSF rep (fetch rep (ram, address rep p)));
(CSF rep (fetch rep (ram, address rep p)));
(FSF rep (fetch rep (ram, address rep p)))
= F)))",
REPEAT GEN_TAC
THEN STRIP_TAC
THEN ASM_REWRITE_TAC[cohn_SPAREFUNC; PAIR_EQ];;

let cohn_sparefunc_TTFF = prove_thm
('cohn_sparefunc_TTFF',
"! (rep:'rep_ty) (a:*wordn) (x:*wordn) (y:*wordn) (p:*wordn) (b:bool).
(((FSF rep (fetch rep (ram, address rep p))) = (T,T,F,F)) ==>
(cohn_SPAREFUNC rep
((DSF rep (fetch rep (ram, address rep p)));
(CSF rep (fetch rep (ram, address rep p)));
(FSF rep (fetch rep (ram, address rep p)))
= F)))",
REPEAT GEN_TAC
THEN STRIP_TAC
THEN ASM_REWRITE_TAC[cohn_SPAREFUNC; PAIR_EQ];;

let cohn_sparefunc_TTFF = prove_thm
('cohn_sparefunc_TTFF',
"! (rep:'rep_ty) (a:*wordn) (x:*wordn) (y:*wordn) (p:*wordn) (b:bool).
(((FSF rep (fetch rep (ram, address rep p))) = (T,T,F,F)) ==>
(cohn_SPAREFUNC rep
((DSF rep (fetch rep (ram, address rep p)));
(CSF rep (fetch rep (ram, address rep p)));
(FSF rep (fetch rep (ram, address rep p)))
= F)))",
REPEAT GEN_TAC
THEN STRIP_TAC
THEN ASM_REWRITE_TAC[cohn_SPAREFUNC; PAIR_EQ];;
% cohn_ILLEGALWRITE_TTFF %
let cohn_ILLEGALWRITE_TTFF = prove_thm
('cohn_ILLEGALWRITE_TTFF',
"@ (rep:'rep_ty) (a:*wordn) (x:*wordn) (y:*wordn) (p:*wordn)
(b:bool) (ram:*memory).
("((DSF rep (fetch rep (ram, address rep p))) = (T,T,T)) \/
((DSF rep (fetch rep (ram, address rep p))) = (T,T,F))) =>
(cohn_ILLEGALWRITE rep (DSF rep
(fetch rep (ram, address rep p)),
(CSF rep (fetch rep (ram, address rep p))),
(MSF rep (fetch rep (ram, address rep p))))
= F)",
REPEAT GEN_TAC THEN STRIP_TAC THEN
IMP_RES_TAC cohn_WRITE_TTFF
THEN ASM_REWRITE_TAC[PAIR_EQ; cohn_ILLEGALWRITE]);;

% cohn_illegalpdest_TTFF_ill %
let cohn_illegalpdest_TTFF_ill = prove_thm
('cohn_illegalpdest_TTFF_ill',
"! (rep:'rep_ty) (a:*wordn) (x:*wordn) (y:*wordn) (p:*wordn)
(b:bool) (ram:*memory).
("((CSF rep (fetch rep (ram, address rep p))) /
(FSF rep (fetch rep (ram, address rep p))) = (T,F,F)) \/
(DSF rep (fetch rep (ram, address rep p))) = (T,F,T)) \/
(DSF rep (fetch rep (ram, address rep p))) = (T,F,T)))
=> ((cohn_ILLEGALPDEST rep (DSF rep
(fetch rep (ram, address rep p)),
(CSF rep (fetch rep (ram, address rep p))),
(FSF rep (fetch rep (ram, address rep p))))
= T)",
REPEAT GEN_TAC
THEN STRIP_TAC
THEN ASM_REWRITE_TAC[PAIR_EQ; cohn_ILLEGALPDEST]);;

% cohn_illegalpdest_TTFF_pass %
let cohn_illegalpdest_TTFF_pass = prove_thm
('cohn_illegalpdest_TTFF_pass',
"@ (rep:'rep_ty) (a:*wordn) (x:*wordn) (y:*wordn) (p:*wordn)
(b:bool) (ram:*memory)."
let cohn_TTFF_FFF_aux = prove_thm
('cohn_TTFF_FFF_aux',
"! (rep:'rep_ty) (a:*wordn) (x:*wordn) (y:*wordn)
(p:*wordn) (b:bool) (stop:bool) (ram:*memory).
(let fsf = (FSF rep (fetch rep (ram, address rep p))) in
let dsf = (DSF rep (fetch rep (ram, address rep p))) in
let msf = (MSF rep (fetch rep (ram, address rep p))) in
let rsf = (RSF rep (fetch rep (ram, address rep p))) in
let csf = (CSF rep (fetch rep (ram, address rep p))) in
let addr = (address rep (fetch rep (ram, address rep p))) in
let newp = (add rep (p, wordn rep 1)) in
let io = ((cohn_OUTPUT rep (dsf, csf)) "/
(cohn_INPUT rep (dsf, csf, fsf))) in
let r = cohn_REG rep (rsf, a, x, y, newp) in
let m = cohn_MEMREAD rep (ram, msf, addr, x, y, io, cohn_NILM rep (dsf, csf, fsf)) in
let aluout = cohn_ALU rep (fsf, msf, dsf, r, m, b) in
let newp = (add rep (p, wordn rep 1)) in
((("stop) "/
("csf) "/
(valid_address rep newp) "/
(\(dsf = (T,T,T))) "/
(\(dsf = (T,T,F))) "/
(\(dsf = (F,F,F))) =>
(fsf = (T,T,F,F)) =>
(cohn_NEXT rep (a, x, y, p, b, F, ram) =
(cohn_VALUE aluout, x, y, newp,
cohn_BVAL aluout, cohn_SVAL aluout, ram))))");
REPEAT GEN_TAC
THEN EXPAND_LET_TAC
THEN STRIP_TAC
THEN (PURE_REWRITE_TAC [cohn_NEXT_expanded]
THEN IMP_RES_TAC cohn_illegalcall_TTFF
THEN IMP_RES_TAC cohn_NILM_TTFF
THEN IMP_RES_TAC cohn_ILLEGALWRITE_TTFF
THEN IMP_RES_TAC cohn_WRITE_TTFF
THEN IMP_RES_TAC cohn_illegalpdest_TTFF_ill
THEN IMP_RES_TAC cohn_illegalpdest~TTFF_pass
THEN IMP_RES_TAC cohn_sparefunc_TTFF
THEN ASM_REWRITE_TAC [cohn_NDOP; cohn_INVALID; cohn_WRITE; cohn_ILLEGALWRITE;
cohn_SPAREFUNC; PAIR_EQ]);;

let cohn_TTFF_FFT_aux = prove_thm
('cohn_TTFF_FFT_aux',
"! (rep:'rep_ty) (a:*wordn) (x:*wordn) (y:*wordn)
(p:*wordn) (b:bool) (stop:bool) (ram:*memory) .

(let fsf = (FSF rep (fetch rep (ram, address rep p))) in
let dsf = (DSF rep (fetch rep (ram, address rep p))) in
let msf = (MSF rep (fetch rep (ram, address rep p))) in
let rsf = (RSF rep (fetch rep (ram, address rep p))) in
let csf = (CSF rep (fetch rep (ram, address rep p))) in
let addr = (address rep (fetch rep (ram, address rep p))) in
let newp = (add rep (p, wordn rep i)) in
let io = ((cohn_OUTPUT rep (dsf, csf)) \/
(cohn_INPUT rep (dsf, csf, fsf))) in

let r = cohn_REG rep (rsf, a, x, y, newp) in
let m = cohn_MEMREAD rep (ram, msf, addr, x,
y, io, cohn_NILM rep (dsf, csf, fsf)) in
let aluout = cohn_ALU rep (fsf, msf, dsf, r, m, b) in
let newp = (add rep (p, wordn rep i)) in

(\((-\text{stop}) /\)
 (\(-\text{csf} /\)
 (\text{valid_address rep newp} /\)
 (\(-\text{dsf} = (T,T,T)) /\)
 (\(-\text{dsf} = (T,F,F)) /\)
 (\text{fsf} = (T,F,F)) ==>
 (cohn_NEXT rep (a, x, y, p, b, F, ram) =
(a, cohn\_VALUE aluout, y, newp, cohn\_BVAL aluout, cohn\_SVAL aluout, ram))))",

96
let cohn_TTFF_FTF_aux = prove_thm
('cohn_TTFF_FTF_aux',
"! (rep:*rep_ty) (a:*wordn) (x:*wordn) (y:*wordn)
(p:*wordn) (b:bool) (stop:bool) (ram:*memory) .
(let fsf = (FSF rep (fetch rep (ram, address rep p))) in
let dsf = (DSF rep (fetch rep (ram, address rep p))) in
let msf = (MSF rep (fetch rep (ram, address rep p))) in
let rsf = (RSF rep (fetch rep (ram, address rep p))) in
let csf = (CSF rep (fetch rep (ram, address rep p))) in
let addr = (address rep (fetch rep (ram, address rep p))) in
let newp = (add rep (p, wordn rep I)) in
let io = ((cohn_OUTPUT rep (dsf, csf)) \/
(cohn_INPUT rep (dsf, csf, fsf))) in
let r = cohn_REG rep (rsf, a, x, y, newp) in
let m = cohn_MEMREAD rep (ram, msf, addr, x, y, io, cohn_ILIMN rep (dsf, csf, fsf)) in
let aluout = cohn_ALU rep (fsf, msf, dsf, r, m, b) in
let newp = (add rep (p, wordn rep 1)) in
(((~stop) \/
  (~csf) \/
  (valid_address rep newp) \/
  (~(dsf = (T,T,T))) \/
  (~(dsf = (T,T,F))) \/
  (dsf = (F,T,F))) =>
  (cohn_Next rep (a, x, y, p, b, F, ram) =
  (a, x, cohn_VALUE aluout, newp, cohn_BVAL aluout, cohn_SVAL aluout,
let cohn_TTFF_FTT_aux = prove_thm
('cohn_TTFF_FTT_aux',

"! (rep:'rep_ty) (a:*wordn) (x:*wordn) (y:*wordn)
(p:*wordn) (b:bool) (stop:bool) (ram:*memory).

(let fsf = (FSF rep (fetch rep Cram, address rep p))) in
let dsf = (DSF rep (fetch rep (ram, address rep p))) in
let msf = (MSF rep (fetch rep (ram, address rep p))) in
let rsf = (RSF rep (fetch rep (ram, address rep p))) in
let csf = (CSF rep (fetch rep (ram, address rep p))) in
let addr = (address rep (fetch rep (ram, address rep p))) in
let newp = (add rep (p, wordn rep 1)) in
let io = (((cohn_OUTPUT rep (dsf, csf)) \/
(cohn_INPUT rep (dsf, csf, fsf))) in
let r = cohn_REG rep (rsf, a, x, y, newp) in
let m = cohn_MEMREAD rep (ram, msf, addr, x, y, io, cohn_NILM rep (dsf, csf, fsf)) in
let aluout = cohn_ALU rep (fsf, msf, dsf, r, m, b) in
let newp = (add rep (p, wordn rep 1)) in

((("stop") \/
("csf") \/
(valid_address rep newp) \/
("(dsf = (T,T,T))") \/
("(dsf = (T,T,F))") \/
(dsf = (F,T,T)) \/
    (fsf = (T,T,F,F))) =>
(cohn_NEXT rep (a, x, y, p, b, F, ram) =
(a, x, y, newp, b, T, ram)))"
let cohn_TTFF_TFF_aux = prove_thm
('cohn_TTFF_TFF_aux',
"! (rep:'rep_ty) (a:*wordn) (x:*wordn) (y:*wordn)
(p:*wordn) (b:bool) (stop:bool) (ram:*memory).
(let fsf = (FSF rep (fetch rep (ram, address rep p))) in
let dsf = (DSF rep (fetch rep (ram, address rep p))) in
let msf = (MSF rep (fetch rep (ram, address rep p))) in
let rsf = (RSF rep (fetch rep (ram, address rep p))) in
let csf = (CSF rep (fetch rep (ram, address.rep p))) in
let addr = (address
rep
(p, wordn rep 1)) in
let r = cohn_REG rep (rsf, a, x, y, newp) in
let m = cohn_MEMREAD rep (ram, msf, addr, x, y, io, cohn NILM rep (dsf, csf, fsf)) in
let aluout = cohn_ALU rep (fsf, msf, dsf, r, m, b) in
let newp = (add rep (p, wordn rep 1)) in
(((~stop) \/
(~csf) \/
(valid_address rep newp) \/
(~(dsf = (T,T,T))) \/
(~(dsf = (T,T,F))) \/
   (dsf = (T,F,F))) \/
   (fsf = (T,T,F,F))) ==> (cohn_NEXT rep (a, x, y, p, b, F, ram) = (a, x, y, newp, b, T, ram))

".
REPEAT GEN_TAC
THEN EXPAND_TAC
THEN STRIP_TAC
THEN (PURE_REWRITE_TAC [cohn_NEXT_expanded]
THEN IMP_RES_TAC cohn_illegalcall_TTFF
THEN IMP_RES_TAC cohn_NILM_TTFF
THEN IMP_RES_TAC cohn_ILLEGALWRITE_TTFF
THEN IMP_RES_TAC cohn_WRITE_TTFF
THEN IMP_RES_TAC cohn_illegalpdest_TTFF
THEN IMP_RES_TAC cohn_illegalpdest_TTFF_pass
THEN IMP_RES_TAC cohn_sparefunc_TTFF
THEN ASM_REWRITE_TAC [cohn_NOOP; cohn_INVALID; cohn_WRITE; cohn_ILLEGALWRITE;
cohn_SPAREFUNC;
PAIR_EQ]));

let cohn_TTFF_TFT_aux = prove_thm
('cohn_TTFF_TFT_aux'.
"! (rep:-rep_ty) (a:*wordn) (x:*wordn) (y:*wordn)
(p:*wordn) (b:bool) (stop:bool) (ram:*memory) .
(let fsf = (FSF rep (fetch rep (ram, address rep p))) in
let dsf = (DSF rep (fetch rep (ram, address rep p))) in
let msf = (MSF rep (fetch rep (ram, address rep p))) in
let rsf = (RSF rep (fetch rep (ram, address rep p))) in
let csf = (CSF rep (fetch rep (ram, address rep p))) in
let addr = (address rep (fetch rep (ram, address rep p))) in
let newp = (add rep (p, wordn rep 1)) in
let io = ((cohn_OUTPUT rep (dsf, csf)) \/
(cohn_INPUT rep (dsf, csf, fsf))) in
let r = cohn_REG rep (rsf, a, x, y, newp) in
let m = cohn_MEMREAD rep (ram, msf, addr, x, y, io, cohn_NILM rep (dsf, csf, fsf)) in
let aluout = cohn_ALU rep (fsf, msf, dsf, r, m, b) in
let newp = (add rep (p, wordn rep 1)) in
((\stop /
(\csf) /
(\valid_address rep newp) /
(\(dsf = (T,T,T))) /
(\(dsf = (T,T,F))) /
(dsf = (T,F,T)) /
(fsf = (T,T,F,F)) =>
(cohn_NEXT rep (a, x, y, p, b, F, ram) =
(a, x, y, newp, b, T, ram))"
REPEAT GEN_TAC
THEN EXPAND_TAC
}
THEN STRIP_TAC
THEN (PURE_REWRITE_TAC [cohn_NEXT_expanded]
THEN IMP_RES_TAC cohnIllegalCall_TTFF
THEN IMP_RES_TAC cohn_WILM_TTFF
THEN IMP_RES_TAC cohn_ILLEGALWRITE_TTFF
THEN IMP_RES_TAC cohn_WRITE_TTFF
THEN IMP_RES_TAC cohn_illegalpdest_TTFF_ill
THEN IMP_RES_TAC cohn_illegalpdest_TTFF_pass
THEN IMP_RES_TAC cohn_sparefunc_TTFF
THEN ASM_REWRITE_TAC 
[cohn_NOOP; cohn_INVkLID; cohn_WRITE; cohn_ILLEGALWRITE;
cohn_SPAREFUNC;
PAIR_EQ]));

quit();

%
let cohn_TTFF_aux = prove_thm
('cohn_TTFF_aux',
"! (rep:'rep_ty) (a:wordn) (x:wordn) (y:wordn)
(p:wordn) (b:bool) (stop:bool) (ram:_memory) .
(let fsf - (FSF rep (fetch rep (ram, address rep p))) in
let dsf = (DSF rep (fetch rep (ram, address rep p))) in
let msf = (MSF rep (fetch rep (ram, address rep p))) in
let rsf = (RSF rep (fetch rep (ram, address rep p))) in
let
csf
= (CSF rep (fetch rep (ram, address rep p))) in
let addr = (address rep (fetch rep (ram, address rep p))) in
let neep = (add rep (p, wordn rep I)) in
let io = ((cohn_OUTPUT rep (dsf, csf)) \/
(cohn_INPUT rep (dsf, csf, fsf))) in
let r = cohn_REG rep (rsf, a, x, y, newp) in
let m = cohn_MEMREAD rep (ram, msf, addr, x, y, io, cohn_NILM rep (dsf, csf, fsf)) in
let aluout = cohn_ALU rep (lsf, msf, dsf, r, m, b) in
let newp = (add rep (p, wordn rep I)) in
(((\"stop) \/
(\"csf) \/
(valid_address rep newp) \/
("(dsf = (T,T,T))) \/
("(dsf = (T,T,F))) \/
((dsf = (F,F,F))) \/
(dsf = (F,F,T)) \/
(dsf = (F,T,F)) \

(dsf = (F,T,T)) \ /
(dsf = (T,F,F)) \ /
(dsf = (T,F,T))) \ /
   (fsf = (T,T,F,F))) =>
(cohn_NEXT rep (a, x, y, p, b, stop, ram) =
  ((dsf = (F,F,F))
   => (cohn_VALUE aluout, x, y, newp,
    cohn_BVAL aluout, cohn_SVAL aluout, ram) |
   ((dsf = (F,F,T))
   => (a, cohn_VALUE aluout, y, newp,
    cohn_BVAL aluout, cohn_SVAL aluout, ram) |
   ((dsf = (F,T,F))
   => (a, x, cohn_VALUE aluout, newp,
    cohn_BVAL aluout, cohn_SVAL aluout, ram) |
   (a, x, y, newp, b, T, ram))))",
REPEAT GEN_TAC
THEN EXPAND_LET_TAC
THEN STRIP_TAC
THEN (PURE_REWRITE_TAC [cohn_NEXT_expanded]
  THEN IMP_RES_TAC cohn_illegalcall_TTFF
  THEN IMP_RES_TAC cohn_HILM_TTFF
  THEN IMP_RES_TAC cohn_ILLEGALWRITE_TTFF
  THEN IMP_RES_TAC cohn_WRITE_TTFF
  THEN IMP_RES_TAC cohn_ILLEGALPDEST_TTFF_ILL
  THEN IMP_RES_TAC cohn_ILLEGALPDEST_TTFF_PASS
  THEN IMP_RES_TAC cohn_sparefunc_TTFF
  THEN ASM_REWRITE_TAC [cohn_NOOP; cohn_INVALID; cohn_WRITE; cohn_ILLEGALWRITE;
    cohn_SPAREFUNC;
    PAIR_EQ]));;
%
%
Author: Tony Leung
University of California, Davis

Prove that the macro level =\Rightarrow cohn level
%
system '/bin/rm cohn_shlb.th';;
set_search_path (search_path() @ lib_dir_list);
loadf 'abstract';;
102
new_theory 'cohn_shlb';
new_parent 'aux_def';
new_parent 'cohn_viper';
new_parent 'macro_def';
new_parent 'cohn_TTFF_aux';
new_parent 'cohn_eqvaux';

let rep_ty = abstract_type 'aux_def' 'opcode';

let cohn_ALU = EXPAND_LET_RULE (definition 'cohn_viper' 'cohn_ALU');
let cohn_SVAL = definition 'cohn_viper' 'cohn_SVAL';
let cohn_BVAL = definition 'cohn_viper' 'cohn_BVAL';
let cohn_VALUE = definition 'cohn_viper' 'cohn_VALUE';
let cohn_INVALID = definition 'cohn_viper' 'cohn_INVALID';
let cohn_ILLEGALCALL = definition 'cohn_viper' 'cohn_ILLEGALCALL';
let cohn_SPAREFUNC = definition 'cohn_viper' 'cohn_SPAREFUNC';
let cohn_ILLEGALPDEST = definition 'cohn_viper' 'cohn_ILLEGALPDEST';
let cohn_WRITE = definition 'cohn_viper' 'cohn_WRITE';
let cohn_ILLEGALWRITE = definition 'cohn_viper' 'cohn_ILLEGALWRITE';
let cohn_NILM = definition 'cohn_viper' 'cohn_NILM';
let cohn_NOOP = definition 'cohn_viper' 'cohn_NOOP';
let cohn_REG = definition 'cohn_viper' 'cohn_REG';

let bt3_remaining_lemma = theorem 'cohn_eqvaux' 'bt3_remaining_lemma';

let reg_eqv = theorem 'cohn_eqvaux' 'reg_eqv';

let cohn_stop = theorem 'cohn_eqvaux' 'cohn_stop';

let cohn_noinc = theorem 'cohn_eqvaux' 'cohn_noinc';

let SHLB = definition 'macro_def' 'SHLB';

let write_reg = EXPAND_LET_RULE (definition 'macro_def' 'write_reg');

let load_r = EXPAND_LET_RULE (definition 'macro_def' 'load_r');

let write_reg_illegalpdest_aux =
  theorem 'cohn_eqvaux' 'write_reg_illegalpdest_aux';

% cohn_ALU_TTFF_TT %
let cohn_ALU_TTFF_TT = prove_thm
  ('cohn_ALU_TTFF_TT',
    (((fsf = (T,T,F,F)) \ (msf = (T,T))) =>
     (let pwrite = (dsf = (F,T,T)) \ ((dsf = (T,F,F)) \ (dsf = (T,F,T)))) in
563
103
**(cohn\_alu rep (fsf, msf, dsf, r, m, b)
= (shlb rep (r, b), (bitn rep r). pwrite)))",
REPEAT GEN_TAC
THEN STRIP_TAC
THEN EXPAND\_LET\_TAC
THEN ASM\_REWRITE\_TAC[cohn\_alu;PAIR\_EQ]);;

% cohn\_alu\_ttff\_tt\_value %
let cohn\_alu\_ttff\_tt\_value = prove\_thm
('cohn\_alu\_ttff\_tt\_value',
"! (rep:'rep\_ty) (fsf:bt4) (msf:bt2)
(dsf:bt3) (r:*wordn) (m:*wordn) (b:bool).
(((fsf = (T,T,F,F)) \ / (msf = (T,T))) ==>
(let pwrite = ((dsf = (F,T,T)) \ / ((dsf = (T,F,F)) \ /
(dsf = (T,F,T))) in
let aluout = cohn\_alu rep (fsf, msf, dsf, r, m, b) in
(cohn\_value aluout = (shlb rep (r,b)))))),
REPEAT GEN_TAC
THEN STRIP_TAC
THEN EXPAND\_LET\_TAC
THEN IMP\_RES\_TAC (EXPAND\_LET\_RULE cohn\_alu\_ttff\_tt)
THEN ASM\_REWRITE\_TAC [cohn\_value]);;

% cohn\_alu\_ttff\_tt\_bval %
let cohn\_alu\_ttff\_tt\_bval = prove\_thm
('cohn\_alu\_ttff\_tt\_bval',
"! (rep:'rep\_ty) (fsf:bt4) (msf:bt2)
(dsf:bt3) (r:*wordn) (m:*wordn) (b:bool).
(((fsf = (T,T,F,F)) \ / (msf = (T,T))) ==>
(let pwrite = ((dsf = (F,T,T)) \ / ((dsf = (T,F,F)) \ /
(dsf = (T,F,T))) in
let aluout = cohn\_alu rep (fsf, msf, dsf, r, m, b) in
(cohn\_bval aluout = (bitn rep r))))",
REPEAT GEN_TAC
THEN STRIP_TAC
THEN EXPAND\_LET\_TAC
THEN IMP\_RES\_TAC (EXPAND\_LET\_RULE cohn\_alu\_ttff\_tt)
THEN ASM\_REWRITE\_TAC [cohn\_bval]);;

% cohn\_alu\_ttff\_tt\_sval %
let cohn\_alu\_ttff\_tt\_sval = prove\_thm
('cohn\_alu\_ttff\_tt\_sval',
"! (rep:'rep\_ty) (fsf:bt4) (msf:bt2)
(dsf:bt3) (r:*wordn) (m:*wordn) (b:bool).

104
let cohn_ALU_TTFF_TT_FFF_SVAL_aux
  = EXPAND_LET_RULE (REWRITE_RULE [PAIR_EQ]
  (SPECL ['"rep:rep_ty"'; "(T,T,F,F)"; "(T,T)"; "(F,F,F)"
  "load_r(rep:rep_ty) (a, x, y, add rep (p, wordn rep 1),
  fetch rep (ram, address rep p))"
  "cohn_MEMREAD (rep:rep_ty) (ram, (T,T), address rep
  (fetch rep (ram, address rep p)), x, y.
  (cohn_OUTPUT rep((F,F,F),F) /
  cohn_INPUT rep((F,F,F),F,T,T,F,F)),
  cohn_NILM rep((F,F,F),F,T,T,F,F))"
  "b:bool"
  cohn_ALU_TTFF_TT_SVAL));

let cohn_ALU_TTFF_TT_FFF_BVAL_aux
  = EXPAND_LET_RULE (REWRITE_RULE [PAIR_EQ]
  (SPECL ['"rep:rep_ty"'; "(T,T,F,F)"; "(T,T)"; "(F,F,F)"
  "load_r(rep:rep_ty) (a, x, y, add rep (p, wordn rep 1),
  fetch rep (ram, address rep p))"
  "cohn_MEMREAD (rep:rep_ty) (ram, (T,T), address rep
  (fetch rep (ram, address rep p)), x, y.
  (cohn_OUTPUT rep((F,F,F),F) /
  cohn_INPUT rep((F,F,F),F,T,T,F,F)),
  cohn_NILM rep((F,F,F),F,T,T,F,F))"
  "b:bool"
  cohn_ALU_TTFF_TT_BVAL));

let cohn_ALU_TTFF_TT_FFF_VALUE_aux
  = EXPAND_LET_RULE (REWRITE_RULE [PAIR_EQ]
  (SPECL ['"rep:rep_ty"'; "(T,T,F,F)"; "(T,T)"; "(F,F,F)"
  "load_r(rep:rep_ty) (a, x, y, add rep (p, wordn rep 1),
  fetch rep (ram, address rep p))"
  "cohn_MEMREAD (rep:rep_ty) (ram, (T,T), address rep
  (fetch rep (ram, address rep p)), x, y.
  (cohn_OUTPUT rep((F,F,F),F) /
  cohn_INPUT rep((F,F,F),F,T,T,F,F)),
  cohn_NILM rep((F,F,F),F,T,T,F,F))"
  "b:bool"
  cohn_ALU_TTFF_TT_BVAL)));
let cohn_ALU_TTFF_TT_SVAL_aux = EXPAND_LET_RULE (REWRITE_RULE [PAIR_EQ] (SPECL ["rep:'rep_ty"; "(T,T,F,F)"; "(T,T)"; "(F,F,T)"; "load_r (rep:'rep_t) (a, x, y, add rep (p, wordn rep I), fetch rep (ram, address rep p))"; "cohn_MEMREAD (rep:'rep_ty) (ram, (T,T), address rep (fetch rep (ram, address rep p)), x, y, (cohn_OUTPUT rep((F,F),F) \/
    cohn_INPUT rep((F,F,F),F,T,T,F,F)),
    cohn_ALU_TTFF_TT_VALUE));;

let cohn_ALU_TTFF_TT_FFT_SVAL_aux = EXPAND_LET_RULE (REWRITE_RULE [PAIR_EQ] (SPECL ["rep:'rep_ty"; "(T,T,F,F)"; "(T,T)"; "(F,F,T)"; "load_r (rep:'rep_t) (a, x, y, add rep (p, wordn rep I), fetch rep (ram, address rep p))"; "cohn_MEMREAD (rep:'rep_ty) (ram, (T,T), address rep (fetch rep (ram, address rep p)), x, y, (cohn_OUTPUT rep((F,F),F) \/
    cohn_INPUT rep((F,F,T),F,T,T,F,F)),
    cohn_ALU_TTFF_TT_SVAL));;

let cohn_ALU_TTFF_TT_FFT_BVAL_aux = EXPAND_LET_RULE (REWRITE_RULE [PAIR_EQ] (SPECL ["rep:'rep_ty"; "(T,T,F,F)"; "(T,T)"; "(F,F,T)"; "load_r (rep:'rep_t) (a, x, y, add rep (p, wordn rep I), fetch rep (ram, address rep p))"; "cohn_MEMREAD (rep:'rep_ty) (ram, (T,T), address rep (fetch rep (ram, address rep p)), x, y, (cohn_OUTPUT rep((F,F),F) \/
    cohn_INPUT rep((F,F,T),F,T,T,F,F)),
    cohn_ALU_TTFF_TT_BVAL));;

let cohn_ALU_TTFF_TT_FFT_VALUE_aux = EXPAND_LET_RULE (REWRITE_RULE [PAIR_EQ] (SPECL ["rep:'rep_ty"; "(T,T,F,F)"; "(T,T)"; "(F,F,T)"; "load_r (rep:'rep_t) (a, x, y, add rep (p, wordn rep I), fetch rep (ram, address rep p))"; "cohn_MEMREAD (rep:'rep_ty) (ram, (T,T), address rep (fetch rep (ram, address rep p)), x, y, (cohn_OUTPUT rep((F,F),F) \/
    cohn_INPUT rep((F,F,T),F,T,T,F,F)),
    cohn_ALU_TTFF_TT_VALUE));;
"cohn_MEMREAD (rep:'rep_ty) (ram, (T,T), address rep
(fetch rep (ram, address rep p)). x, y,
(cohn_OUTPUT rep((F,F,T),F) \/
cohn_NILM rep((F,F,T),F,T,T,F,F))";
"b:bool"
cohn_ALU_TTFF_TT_VALUE);;

let cohn_ALU_TTFF_TT_FTF_SVAL_aux
= EXPAND_LET_RULE (REWRITE_RULE [PAIR_EQ]
(SPECL ['"rep:'rep_ty"; "(T,T,F,F)"; "(T,T)"; "(F,T,F)";
"load_r (rep:'rep_ty) (a, x, y, add rep (p, wordn rep 1),
fetch rep (ram, address rep p))";
"cohn_MEMREAD (rep:'rep_ty) (ram, (T,T), address rep
(fetch rep (ram, address rep p)). x, y,
(cohn_OUTPUT rep((F,F,F),F) \/
cohn_NILM rep((F,F,F),F,T,T,F,F))";
"b:bool"
cohn_ALU_TTFF_TT_SVAL));;

let cohn_ALU_TTFF_TT_FTF_BVAL_aux
= EXPAND_LET_RULE (REWRITE_RULE [PAIR_EQ]
(SPECL ['"rep:'rep_ty"; "(T,T,F,F)"; "(T,T)"; "(F,T,F)";
"load_r (rep:'rep_ty) (a, x, y, add rep (p, wordn rep 1),
fetch rep (ram, address rep p))";
"cohn_MEMREAD (rep:'rep_ty) (ram, (T,T), address rep
(fetch rep (ram, address rep p)). x, y,
(cohn_OUTPUT rep((F,F,F),F) \/
cohn_NILM rep((F,F,F),F,T,T,F,F))";
"b:bool"
cohn_ALU_TTFF_TT_BVAL));;

let cohn_ALU_TTFF_TT_FTF_VALUE_aux
= EXPAND_LET_RULE (REWRITE_RULE [PAIR_EQ]
(SPECL ['"rep:'rep_ty"; "(T,T,F,F)"; "(T,T)"; "(F,T,F)";
"load_r (rep:'rep_ty) (a, x, y, add rep (p, wordn rep 1),
fetch rep (ram, address rep p))";
"cohn_MEMREAD (rep:'rep_ty) (ram, (T,T), address rep
(fetch rep (ram, address rep p)), x, y,
(cohn_OUTPUT rep((F,T,F),F) \/
cohn_INPUT rep((F,T,F),F,T,T,F,F)),
cohn_NILM rep((F,T,F),F,T,T,F,F))
"b:bool"
cohn_ALU_TTFF_TT_VALUE));;

let illegal_shlb = (SPECL ["rep:'rep__y";
"a: _vordlt" ;
"x:_ordn' ;
"y :weordxl" ;
"add (rep:'rep_ty) (p,
wordn
rep
I)";
"b:bool"; "F";
"fetch (rep:'rep_ty) (ram, address rep p)";
"ram:*memory";
"shlb (rep:'rep_ty)
((load_r rep
(a,x,y,add rep(p,wordn rep 1),
fetch rep(ram,address rep p))), b)";
"b:bool"]
write_reg_illegalpdest_aux));

let dsf_remain = (SPEC ((DSF rep:'rep_ty)
(fetch rep (ram, address rep p))):bt3
bt3_remaining_lemma));;

let cohn_TTFF_FFF_aux_expanded = EXPAND_LET_RULE
(theorem 'cohn_TTFF_aux 'cohn_TTFF_FFF_aux');;
let cohn_TTFF_FFT_aux_expanded = EXPAND_LET_RULE
(theorem 'cohn_TTFF_aux 'cohn_TTFF_FFT_aux');;
let cohn_TTFF_FTF_aux_expanded = EXPAND_LET_RULE
(theorem 'cohn_TTFF_aux 'cohn_TTFF_FTF_aux');;
let cohn_TTFF_FTT_aux_expanded = EXPAND_LET_RULE
(theorem 'cohn_TTFF_aux 'cohn_TTFF_FTT_aux');;
let cohn_TTFF_TFF_aux_expanded = EXPAND_LET_RULE
(theorem 'cohn_TTFF_aux 'cohn_TTFF_TFF_aux');;
let cohn_TTFF_TFT_aux_expanded = EXPAND_LET_RULE
(theorem 'cohn_TTFF_aux 'cohn_TTFF_TFT_aux');;

% shlb %

set_goal([].
"! (rep:'rep_ty) (a:*wordn) (x:*wordn) (y:*wordn) (p:*wordn) (b:bool)
(stop:bool) (ram:*memory) .
((CSF rep (fetch rep (ram, (address rep p)))) \/
 ((DSF rep (fetch rep (ram, (address rep p))))=(T,T,F)) \/
 ((DSF rep (fetch rep (ram, address rep p)))=(T,T,F)) \/

108
(FSF rep (fetch rep (ram, address rep p)) = (T,T,F,F)) /
(MSF rep (fetch rep (ram, address rep p)) = (T,T)) =>
(SHLB rep (a, x, y, p, b, stop, ram) =
cohn_NEXT rep (a, x, y, p, b, stop, ram)));;

e (REPEAT GEN_TAC
THEN STRIP_TAC
THEN PURE_REWRITE_TAC[SHLB]
THEN EXPAND_LET_TAC
THEN ASM_CASES_TAC "stop : bool"
THEN IMP_RES_TAC cohn_stop
THEN ASM_REWRITE_TAC[]
THEN ASM_CASES_TAC "((DSF (rep : rep_ty) (fetch rep (ram, address rep p))) ffi (F,F,F)) : bool");;

e (IMP_RES_TAC (EXPAND_LET_RULE cohn_nolnc)
THEN ASM_REWRITE_TAC[]
THEN ASM_REWRITE_TAC[]);

e (ASSUM_LIST (\asl. ASSUME_TAC (REWRITE_RULE
[el 19 asl] (el 1 asl)))
THEN ASM_REWRITE_TAC[]);

e (ASM_CASES_TAC "((DSF (rep : rep_ty) (fetch rep (ram, address rep p)))
= (F,F,F)) : bool");;

e (IMP_RES_TAC cohn_TTFF_FFF_aux_expanded
THEN ASM_REWRITE_TAC [cohn_ALU_TTFF_TT_FFF_VALUE_aux;
cohn_ALU_TTFF_TT_FFF_SVAL_aux;
cohn_ALU_TTFF_TT_FFF_BVAL_aux;
reg_eqv; write_reg; PAIR_EQ]));;

e (ASM_CASES_TAC "((DSF (rep : rep_ty) (fetch rep (ram, address rep p)))
= (F,F,T)) : bool");;

e (IMP_RES_TAC cohn_TTFF_FFT_aux_expanded
THEN ASM_REWRITE_TAC [cohn_ALU_TTFF_TT_FFT_VALUE_aux;
cohn_ALU_TTFF_TT_FFT_SVAL_aux;
cohn_ALU_TTFF_TT_FFT_BVAL_aux;
reg_eqv; write_reg; PAIR_EQ]]);;

e (ASM_CASES_TAC "((DSF (rep : rep_ty) (fetch rep (ram, address rep p)))
= (F,T,F)) : bool");;

e (IMP_RES_TAC cohn_TTFF_FTF_aux_expanded
THEN ASM_REWRITE_TAC [cohn_ALU_TTFF_TT_FTF_VALUE_aux;
cohn_ALU_TTFF_TT_FTF_SVAL_aux;
cohn_ALU_TTFF_TT_FTF_BVAL_aux;
reg_eqv; write_reg; PAIR_EQ])];;

e (ASM_CASES_TAC "((DSF (rep : rep_ty) (fetch rep (ram, address rep p)))
= (F,F,T)) : bool");;
colm_ALU_TTFF_TT_FTF_SVAL_aux;
cohn_ALU_TTFF_TT_FTF_BVAL_aux;
reg_eqv; write_reg; PAIR_EQ);;

e (ASM_CASES_TAC "((DSF (rep:'rep_ty) (fetch rep (ram, address rep p)))
  = (F,T,T)):bool");;
e (IMP_RES_TAC coln_TTFF_FTT_aux_expanded
  THEN IMP_RES_TAC illegal_shlb
  THEN ASM_REWRITE_TAC [reg_eqv; write_reg; PAIR_EQ]);;
e (ASM_CASES_TAC "((DSF (rep:'rep_ty) (fetch rep (ram, address rep p)))
  = (T,F,F)):bool");;
e (IMP_RES_TAC coln_TTFF_TFF_aux_expanded
  THEN IMP_RES_TAC illegal_shlb
  THEN ASM_REWRITE_TAC [reg_eqv; write_reg; PAIR_EQ]);;
e (IMP_RES_TAC dsf~remain);

e (IMP_RES_TAC coln_TTFF_TFF_aux_expanded
  THEN IMP_RES_TAC illegal_shlb
  THEN ASM_REWRITE_TAC [reg_eqv; write_reg; PAIR_EQ]);;
Appendix D: MACRO LEVEL SPECIFICATION

%---------------------------------------------------------------

File: def_ucode.ml

Description: Defines the selectors for fields of a microinstruction

---------------------------------------------------------------% 

set_search_path (search_path() @ lib_dir_list);;

system '/bin/rm ucode_def.th';;

new_theory 'ucode_def';;

map new_parent ['tuple'];;

%---------------------------------------------------------------

Now define a type for ucode.

---------------------------------------------------------------% 

new_type_abbrev('ucode',

":(bt7#(bt3#bt4)#bool#(bool#bool#bool)#(bt2#bt3#bool#bool)#
  (bool#bool)#(bool#bool#bt2))"
);

%---------------------------------------------------------------

Here are the selectors for the microcode

---------------------------------------------------------------% 

let Maddr = new_definition

('Maddr',

"!(rd wr inout decctl rfctl dfctl den ren asel dsel:bool) 
  Maddr (ua,(sctl,actl),decctl,(rd,wr,inout),(urf,udf,rfctl,dfctl),
  (den,ren), (asel,dsel,msel)) = ua"
);

let Seqctl = new_definition

('Seqctl',

"!(rd wr inout decctl rfctl dfctl den ren asel dsel:bool) 
  Seqctl (ua,(sctl,actl),decctl,(rd,wr,inout),(urf,udf,rfctl,dfctl),
  (den,ren), (asel,dsel,msel)) = sctl"
);

let Aluctl = new_definition

('Aluctl',

"!(rd wr inout decctl rfctl dfctl den ren asel dsel:bool) 

let Dec_ctl = new_definition
('Dec_ctl',
  '!(rd wr inout decctl rfctl dfctl den ren asel dsel:bool)
  Dec_ctl (ua,(sctl,actl),decctl,(rd,wr,inout),(urf,udf,rfctl,dfctl),
    (den,ren), (asel,dsel,msel)) = decctl''
);

let R = new_definition
('R',
  '!(rd wr inout decctl rfctl dfctl den ren asel dsel:bool)
  (urf msel:bt2) (actl:bt4) (ua:bt7).
  R (ua,(sctl,actl),decctl,(rd,wr,inout),(urf,udf,rfctl,dfctl),
    (den,ren), (asel,dsel,msel)) = rd''
);

let W = new_definition
('W',
  '!(rd wr inout decctl rfctl dfctl den ren asel dsel:bool)
  W (ua,(sctl,actl),decctl,(rd,wr,inout),(urf,udf,rfctl,dfctl),
    (den,ren), (asel,dsel,msel)) = wr''
);

let Io = new_definition
('Io',
  '!(rd wr inout decctl rfctl dfctl den ren asel dsel:bool)
  Io (ua,(sctl,actl),decctl,(rd,wr,inout),(urf,udf,rfctl,dfctl),
    (den,ren), (asel,dsel,msel)) = inout''
);

let Mrf = new_definition
('Mrf',
  '!(rd wr inout decctl rfctl dfctl den ren asel dsel:bool)
  Mrf (ua,(sctl,actl),decctl,(rd,wr,inout),(urf,udf,rfctl,dfctl),
    (den,ren), (asel,dsel,msel)) = urf''
);

let Mdf = new_definition
('Mdf',
  '!(rd wr inout decctl rfctl dfctl den ren asel dsel:bool)
  (urf msel:bt2) (actl:bt4) (ua:bt7).
  Mdf (ua,(sctl,actl),decctl,(rd,wr,inout),(urf,udf,rfctl,dfctl),
    (den,ren), (asel,dsel,msel)) = urf''
);
let Rfc = new_definition
('Rfc',
"!(rd wr inout decctl rfctl dfctl den ren asel dsel:bool)
Rfc (ua,(sctl,actl),decctl,(rd,wr,inout),(urf,udf,rfctl,dfctl),
 (den,ren), (asel,dsel,msel)) = rfctl"
);

let Dfc = new_definition
('Dfc',
"!(rd wr inout decctl rfctl dfctl den ren asel dsel:bool)
Dfc (ua,(sctl,actl),decctl,(rd,wr,inout),(urf,udf,rfctl,dfctl),
 (den,ren), (asel,dsel,msel)) = dfctl"
);

let De = new_definition
('De',
"!(rd wr inout decctl rfctl dfctl den ren asel dsel:bool)
De (ua,(sctl,actl),decctl,(rd,wr,inout),(urf,udf,rfctl,dfctl),
 (den,ren), (asel,dsel,msel)) = den"
);

let Re = new_definition
('Re',
"!(rd wr inout decctl rfctl dfctl den ren asel dsel:bool)
Re (ua,(sctl,actl),decctl,(rd,wr,inout),(urf,udf,rfctl,dfctl),
 (den,ren), (asel,dsel,msel)) = ren"
);

let Adrs = new_definition
('Adrs',
"!(rd wr inout decctl rfctl dfctl den ren asel dsel:bool)
Adrs (ua,(sctl,actl),decctl,(rd,wr,inout),(urf,udf,rfctl,dfctl),
 (den,ren), (asel,dsel,msel)) = asel"
);

let Ds = new_definition
('Ds',
"!(rd wr inout decctl rfctl dfctl den ren asel dsel:bool)
Ds (ua,(sctl,actl),decctl,(rd,wr,inout),(urf,udf,rfctl,dfctl),
 (den,ren), (asel,dsel,msel)) = dsel"
);

let Mdf (ua,(actl,actl),decctl,(rd,wr,inout),(urf,udf,rfctl,dfctl),
 (den,ren), (asel,dsel,msel)) = udf"
"!(rd wr inout decctl rfctl dfctl den ren asel dsel:bool)
Ds (ua,(sctl,actl),decctl,(rd,wr,inout),(urf,udf,rfctl,dfctl),
(den,ren), (asel,dsel,msel)) = dsel"
);

let Ms = new_definition
('Ms',
"!(rd wr inout decctl rfctl dfctl den ren asel dsel:bool)
Ms (ua,(sctl,actl),decctl,(rd,wr,inout),(urf,udf,rfctl,dfctl),
(den,ren), (asel,dsel,msel)) = msel"
);

close_theory();
The Macro level of Viper

University of California, Davis

Viper's macro level

modifications

- changed formatting and reordered opnds to add rep
- SUBS changed to SUBO
- changed write_preg so that if skip, stop is set to F
- changed SHL to use ovflw = bitn rep ldr
- added expanded defns for load_m, load_r, etc

% system '/bin/rm macro_def.th';;

set_search_path (search_path() @ lib_dir_list);;

loadf 'abstract';;

new_theory 'macro_def';;

new_parent 'aux_def';;

let rep_ty = abstract_type 'aux_def' 'opcode';;

%---------------------------------------------------------------

start of addressing unit

---------------------------------------------------------------%

let load_m = new_definition('load_m'),

"! (rep:rep ty) (a:*wordn) (x:*wordn) (y:*wordn) (p:*wordn)
 (ir:*wordn) (ram:*memory) .
load_m rep (a, x, y, p, ir, ram) =
let msfValue = (MSF rep ir) in
let tmp = (address rep ir) in
let addr = (pad rep tmp) in
((msfValue = (F,F)) => (F, addr) |
 ((msfValue = (F,T)) => (F, (fetch rep (ram, (address rep addr))))) |
 ((msfValue = (T,F)) => (let t = (add rep (x, addr)) in
 ((valid_address rep t) =>
 (F, fetch rep (ram, (address rep t)))) |
 (T, addr)) |
 (let t = (add rep (y, addr)) in
 ((valid_address rep t) =>
 ...
(F, (fetch rep (ram, (address rep t))) | 
(T, addr))));;

save_thm('load_m_expanded', EXPAND_LET_RULE load_m);

let load_io = new_definition('load_io', 
"! (rep:'rep_ty) (a:*wordn) (x:*wordn) (y:*wordn) (p:*wordn) 
(ir:*wordn) (ram:*memory) .
load_io rep a x y p ir ram =
let msfValue = (MSF rep ir) in
let tmp = (address rep ir) in
let addr = (pad rep tmp) in
((msfValue = (F,F)) => (F, addr) | 
((msfValue = (F,T)) => (F, (fetchio rep (ram, (address rep addr)))) | 
((msfValue = (T,F)) => (let t = (add rep (x, addr)) in 
((valid_address rep t) => 
(F, fetchio rep (ram, (address rep t)))) | 
(T, addr)))));

save_thm('load_io_expanded', EXPAND_LET_RULE load_io);

let load_r = new_definition('load_r', 
"! (rep:'rep_ty) (a:*wordn) (x:*wordn) (y:*wordn) (p:*wordn) 
(ir:*wordn) .
load_r rep a x y p ir =
let rsfValue = (RSF rep ir) in
((rsfValue = (F,F)) => a | 
((rsfValue = (F,T)) => x | 
((rsfValue = (T,F)) => y | 
p))));;

save_thm('load_r_expanded', EXPAND_LET_RULE load_r);

let write_reg = new_definition('write_reg', 
"! (rep:'rep_ty) (a:*wordn) (x:*wordn) (y:*wordn) (p:*wordn) (b:bool) 
(stop:bool) (ir:*wordn) (ram:*memory) (value:*wordn) (newb:bool).
write_reg rep a x y p b stop ir ram value newb = 
let dsfValue = (DSF rep ir) in 
((dsfValue = (F,F,F)) => (value, x, y, p, newb, stop, ram) | 
(F, (fetch rep (ram, (address rep t)))) | 
(T, addr))));;

save_thm('write_reg_expanded', EXPAND_LET_RULE write_reg);
let write_preg = new_definition('write_preg',
"! (rep:*rep_ty) (a:*wordn) (x:*wordn) (y:*wordn) (p:*wordn) (b:bool)
  (stop:bool) (ir:*wordn) (ram:*memory) (value:*wordn).
write_preg rep (a, x, y, p, b, stop, ir, ram, value) =
let dsfValue = (DSF ir) in
let call = ((CSF rep Jr) = (F)) /
  ((FSF rep Jr) = (F,F,T)) in
((dsfValue = (F,F,F)) => (value, x, y, p, b, stop, ram) |
((dsfValue = (F,F,T)) => (a, value, y, p, b, stop, ram) |
((dsfValue = (F,T,F)) => (a, value, p, b, stop, ram) |
((dsfValue = (T,F,F)) => (a, x, y, p, b, T, ram) |
((dsfValue = (T,T,F)) => (a, x, y, b, T, ram) |
((dsfValue = (T,T,T)) => (a, x, y, p, b, T, ram) |
((dsfValue = (T,F,F)) /
  ((dsfValue = (T,T,F)) / b)) => (a, x, y, p, b, F, ram) |
(call => (a, x, p, value, b, (("valid_address rep value") \ stop),
  ram) | (a, x, y, value, b, (stop\"valid_address rep value\), ram))))));

save_thm('write_preg_expanded', EXPAND_LET_RULE write_preg);

%------------------------------------------------------------------
% start of ALU
%------------------------------------------------------------------
%
let CMP = new_definition('CMP',
"! (rep:*rep_ty) (a:*wordn) (x:*wordn) (y:*wordn) (p:*wordn) (b:bool)
  (stop:bool) (ram:*memory).
CMP rep (a, x, y, p, b, stop, ram) =
(stop => (a, x, y, p, b, stop, ram) |
(let newp = (add rep (p, wordn rep 1)) in
("valid_address rep newp") =>
(a, x, y, newp, b, T, ram) |
(let ir = (fetch rep (ram, address rep p)) in
let m = (load_m rep (a, x, y, newp, ir, ram)) in
((FST m) => % invalid memory load %
(a, x, y, newp, b, T, ram) |
(let ldr = (load_r rep (a, x, y, newp, ir)) in
let ldm = (SND m) in
let fsf = (FSF rep ir) in
(a, x, y, newp, (bcmp rep ldr, ldm, b, fsf), F, ram)))

% negate a value
%
let NEG = new_definition('NEG',
"! (rep:'rep_ty) (a:*wordn) (x:*wordn) (y:*wordn) (p:*wordn)
   (b:bool) (stop:bool) (ram:*memory).
   NEG rep (a, x, y, p, b, stop, ram) =
   (stop => (a, x, y, p, b, stop, ram)) |
   (let newp = (add rep (p, wordn rep i)) in
    ("valid_address rep newp) =>
    (a, x, y, newp, b, T, ram) |
   (let ir = (fetch rep (ram, address rep p)) in
    let m = (load_m rep (a, x, y, newp, ir, ram)) in
    ((FST m) => invalid memory load %
    (a, x, y, newp, b, T, ram) |
   (let ldm = (SND m) in
    let result = (neg rep ldm) in
    write_reg rep (a, x, y, newp, b, F, ir, ram; result, b))))

%
Add without overflow detection.
%
let ADDB = new_definition('ADDB',
"! (rep:'rep_ty) (a:*wordn) (x:*wordn) (y:*wordn) (p:*wordn)
   (b:bool) (stop:bool) (ram:*memory).
   ADDB rep (a, x, y, p, b, stop, ram) =
   (stop => (a, x, y, p, b, stop, ram)) |
   (let newp = (add rep (p, wordn rep i)) in
    ("valid_address rep newp) =>
    (a, x, y, newp, b, T, ram) |
   (let ir = (fetch rep (ram, address rep p)) in
    let m = (load_m rep (a, x, y, newp, ir, ram)) in
    ((FST m) => invalid memory load %
    (a, x, y, newp, b, T, ram) |
   (let ldm = (SND m) in
    let ldr = (load_r rep (a, x, y, newp, ir)) in
    let result = (add rep ldr, ldm) in
    get result, addition with carry %

118
let carry = (addp rep (ldr, ldm, result)) in % get carry %
write_reg rep (a, x, y, newp, b,F,ir,ram,result,carry))))));

%
Add with overflow detection.
%
let ADDS = new_definition('ADDS',
"! (rep:'rep_ty) (a:*wordn) (x:*wordn) (y:*wordn) (p:*wordn) (b:bool) (stop:bool) (ram:*memory) .
ADDS rep (a, x, y, p, b, stop, ram) =
(stop => (a, x, y, p, b, stop, ram) |
(let newp = (add rep (p, wordn rep 1)) in
("valid_address rep newp) =>
(a, x, y, newp, b, T, ram) |
(let ir = (fetch rep (ram, address rep p)) in
let m = (load_m rep (a, x, y, newp, ir, ram)) in
((FST m) => % invalid memory load %
(a, x, y, newp, b, T, ram) |
(let ld = (SND m) in
let ldr = (load_r rep (a, x, y, newp, ir)) in
let result = (add rep (ldr, ld)) in
let ovflw = (aovfl rep (ldr, ld, result)) in % detect overflow%
write_reg rep (a, x, y, newp, b, ovflw, ir, ram, result))))))));

%
Subtract without overflow detection.
%
let SUBB = new_definition('SUBB',
"! (rep:'rep_ty) (a:*wordn) (x:*wordn) (y:*wordn) (p:*wordn) (b:bool) (stop:bool) (ram:*memory) .
SUBB rep (a, x, y, p, b, stop, ram) =
(stop => (a, x, y, p, b, stop, ram) |
(let newp = (add rep (p, wordn rep 1)) in
("valid_address rep newp) =>
(a, x, y, newp, b, T, ram) |
(let ir = (fetch rep (ram, address rep p)) in
let m = (load_m rep (a, x, y, newp, ir, ram)) in
((FST m) => % invalid memory load %
(a, x, y, newp, b, T, ram) |
(let ld = (SND m) in
let ldr = (load_r rep (a, x, y, newp, ir)) in
let result = (sub rep (ldr, ld)) in
let carry = (subp rep (ldr, ld, result)) in % detect carry %
write_reg rep (a, x, y, newp, b, F, ir, ram,result,carry))))))));
% Subtract with overflow detection
%
let SUBO = new_definition('SUBO',
  "! (rep:*rep_ty) (a:*wordn) (x:*wordn) (y:*wordn) (p:*wordn) (b:bool) (stop:bool) (ram:*memory)
SUBO rep (a, x, y, p, b, stop, ram) =
(stop => (a, x, y, p, b, stop, ram) | (let newp = (add rep (p, wordn rep 1)) in
("valid_address rep newp) =>
(a, x, y, newp, b, T, ram) | (let ir = (fetch rep (ram, address rep p)) in
  let m = (load_m rep (a, x, y, newp, ir, ram)) in
((FST m) => % invalid memory load %
(a, x, y, newp, b, T, ram) | (let ldm = (SND m) in
  let ldr = (load_r rep (a, x, y, newp, ir)) in
    let result = (sub rep (ldr, ldm)) in
    let ovflw = (sovfl rep (ldr, ldm, result)) in % overflow detection %
write_reg rep (a, x, y, newp, b, ovflw, ir, ram, result))))))");;

% Exclusive OR between two operands
%
let XOR = new_definition('XOR',
  "! (rep:*rep_ty) (a:*wordn) (x:*wordn) (y:*wordn) (p:*wordn) (b:bool) (stop:bool) (ram:*memory)
  XOR rep (a, x, y, p, b, stop, ram) =
(stop => (a, x, y, p, b, stop, ram) | (let newp = (add rep (p, wordn rep 1)) in
("valid_address rep newp) =>
(a, x, y, newp, b, T, ram) | (let ir = (fetch rep (ram, address rep p)) in
  let m = (load_m rep (a, x, y, newp, ir, ram)) in
((FST m) => % invalid memory load %
(a, x, y, newp, b, T, ram) | (let ldm = (SND m) in
  let ldr = (load_r rep (a, x, y, newp, ir)) in
    let result = (bxor rep (ldr, ldm)) in
write_reg rep (a, x, y, newp, b, F, ir, ram, result, b)))))");;

% And between two operands
%

let AND = new_definition('AND',
"! (rep:rep_ty) (a:*wordn) (x:*wordn) (y:*wordn) (p:*wordn) (b:bool)
  (stop:bool) (ram:*memory) .
AND rep (a, x, y, p, b, stop, ram) =
  (stop => (a, x, y, p, b, stop, ram)) |
  (let newp = (add rep (p, wordn rep 1)) in
   ("valid_address rep newp") =>
    (a, x, y, newp, b, T, ram) |
    (let ir = (fetch rep (ram, address rep p)) in
     let m = (load_m rep (a, x, y, newp, ir, ram)) in
     ((FST m) => % invalid memory load %
      (a, x, y, newp, b, T, ram) |)
    (let ldm = (SND m) in
     let ldr = (load_r rep (a, x, y, newp)) in
     let result = (band rep (ldr, ldm)) in
     write_reg rep (a, x, y, newp, b, F, ir, ram, result, b)))));;
%
NOR between two operands
%
let NOR = new_definition('NOR',
"! (rep:rep_ty) (a:*wordn) (x:*wordn) (y:*wordn) (p:*wordn) Cb:bool)
  (stop:bool) (ram:*memory) .
NOR rep (a, x, y, p, b, stop, ram) =
  (stop => (a, x, y, p, b, stop, ram)) |
  (let newp = (add rep (p, wordn rep 1)) in
   ("valid_address rep newp") =>
    (a, x, y, newp, b, T, ram) |)
  (let ir = (fetch rep (ram, address rep p)) in
   let m = (load_m rep (a, x, y, newp, ir, ram)) in
   ((FST m) => % invalid memory load %
    (a, x, y, newp, b, T, ram) |
    (let ldm = (SND m) in
     let ldr = (load_r rep (a, x, y, newp, ir)) in
     let result = (bnor rep (ldr, ldm)) in
     write_reg rep (a, x, y, newp, b, F, ir, ram, result, b))))));;
%
ANDMBAR between two operands
%
let ANDMBAR = new_definition('ANDMBAR',
"! (rep:rep_ty) (a:*wordn) (x:*wordn) (y:*wordn) (p:*wordn) (b:bool)
  (stop:bool) (ram:*memory) .
ANDMBAR rep (a, x, y, p, b, stop, ram) =
  (stop => (a, x, y, p, b, stop, ram)) |
(let newp = (add rep (p, wordn rep 1)) in
  ('valid_address rep newp) =>
  (a, x, y, newp, b, T, ram) |
  (let ir = (fetch rep (ram, address rep p)) in
    let m = (load_m rep (a, x, y, newp, ir, ram)) in
    ((FST m) => % invalid memory load %
      (a, x, y, newp, b, T, ram) |)
  (let ldm = (SND m) in
    let ldr = (load_r rep (a, x, y, newp, ir)) in
    let result = (band rep (ldr, bnot rep ldm)) in
      write_reg rep (a, x, y, newp, b, F, ir, ram, result, b))))) ;;

% Shift right, copy sign bit
%
let SHR = new_definition('SHR',
  "! (rep:'rep_ty) (a:*wordn) (x:*wordn) (y:*wordn) (p:*wordn) (b:bool)
   (stop:bool) (ram:*memory) .
   SHR rep (a, x, y, p, b, stop, ram) =
   (stop => (a, x, y, p, b, stop, ram) |)
   (let newp = (add rep (p, wordn rep 1)) in
    ('valid_address rep newp) =>
      (a, x, y, newp, b, T, ram) |)
  (let ir = (fetch rep (ram, address rep p)) in
    let ldr = (load_r rep (a, x, y, newp, ir)) in
    let result = (shrb rep (Idr. b)) in
      let newb = (bitO rep idr) in
        write_reg rep (a, x, y, newp, b, F, ir, ram, result, newb))))) ;;
%
Shift right through b
%
let SHRB = new_definition('SHRB',
  "! (rep:'rep_ty) (a:*wordn) (x:*wordn) (y:*wordn) (p:*wordn) (b:bool)
   (stop:bool) (ram:*memory) .
   SHRB rep (a, x, y, p, b, stop, ram) =
   (stop => (a, x, y, p, b, stop, ram) |)
   (let newp = (add rep (p, wordn rep 1)) in
    ('valid_address rep newp) =>
      (a, x, y, newp, b, T, ram) |)
  (let ir = (fetch rep (ram, address rep p)) in
    let ldr = (load_r rep (a, x, y, newp, ir)) in
    let result = (shrb rep ldr) in
      write_reg rep (a, x, y, newp, b, F, ir, ram, result, b))))) ;;

122
% Shift left
%
let SHL = new_definition('SHL',
  "! (rep:'rep_ty) (a:*wordn) (x:*wordn) (y:*wordn) (p:*wordn) (b:bool)
    (stop:bool) (ram:*memory) .
SHL rep (a, x, y, p, b, stop, ram) =
  (stop => (a, x, y, p, b, stop, ram) |
  (let newp = (add rep (p, wordn rep 1)) in
   ('valid_address rep newp) =>
    (a, x, y, newp, b, T, ram) |
    (let ir = (fetch rep (ram, address rep p)) in
     let ldr = (load_r rep (a, x, y, newp, ir)) in
     let result = (shl rep ldr) in
     % let ovflw = (aovfl rep (ldr, ldr, result)) in
     let ovflw = (bitn rep ir) in
     write_reg rep (a, x, y, newp, b, ovflw, ir, ram, result, b)))))");;
%
Shift left through b
%
let SHLB = new_definition('SHLB',
  "! (rep:'rep_ty) (a:*wordn) (x:*wordn) (y:*wordn) (p:*wordn) (b:bool)
    (stop:bool) (ram:*memory) .
SHLB rep (a, x, y, p, b, stop, ram) =
  (stop => (a, x, y, p, b, stop, ram) |
  (let newp = (add rep (p, wordn rep 1)) in
   ('valid_address rep newp) =>
    (a, x, y, newp, b, T, ram) |
    (let ir = (fetch rep (ram, address rep p)) in
     let ldr = (load_r rep (a, x, y, newp, ir)) in
     let result = (shlb rep ldr) in
     let newb = (bitn rep ir) in
     write_reg rep (a, x, y, newp, b, F, ir, ram, result, newb)))))");;
%
let CALL = new_definition('CALL',
  "! (rep:'rep_ty) (a:*wordn) (x:*wordn) (y:*wordn) (p:*wordn) (b:bool)
    (stop:bool) (ram:*memory) .
CALL rep (a, x, y, p, b, stop, ram) =
  (stop => (a, x, y, p, b, stop, ram) |
  (let newp = (add rep (p, wordn rep 1)) in
   ('valid_address rep newp) =>
    let ovflw = (aovfl rep (ldr, ldr, result)) in
    write_reg rep (a, x, y, newp, b, F, ir, ram, result, newb)))");;
(a, x, y, newp, b, T, ram) |
(let ir = (fetch rep (ram, address rep p)) in
 let m = (load_m rep (a, x, y, newp, ir, ram)) in
((FST m) => % invalid memory load %
(a, x, y, newp, b, T, ram) |
(let ldm = (SND m) in
 let ldr = (load_r rep (a, x, y, newp, ir)) in
 let dsf = (DSF rep ir) in
 (a, x, newp, ldm, b, (valid_address rep ldm), ram)))))));

% was: write_reg rep (a, x, y, newp, b, F, ir, ram, ldm)))})));

let READM = new_definition('READM',
"! (rep:rep_ty) (a:*wordn) (x:*wordn) (y:*wordn) (p:*wordn) (b:bool)
 (stop:bool) (ram:*memory) .
 READM rep (a, x, y, p, b, stop, ram) =
 (stop => (a, x, y, p, b, stop, ram) |
(let newp = (add rep (p, wordn rep p)) in
 (valid_address rep newp) =>
 (a, x, y, newp, b, T, ram) | 
(let ir = (fetch rep (ram, address rep p)) in
 let m = (load_m rep (a, x, y, newp, ir, ram)) in
((FST m) => % invalid memory load %
(a, x, y, newp, b, T, ram) |
(let ldm = (SND m) in
 write_reg rep (a, x, y, newp, b, F, ir, ram, ldm))))))));

let READIO = new_definition('READIO',
"! (rep:rep_ty) (a:*wordn) (x:*wordn) (y:*wordn) (p:*wordn) (b:bool)
 (stop:bool) (ram:*memory) .
 READIO rep (a, x, y, p, b, stop, ram) =
 (stop => (a, x, y, p, b, stop, ram) |
(let newp = (add rep (p, wordn rep p)) in
 (valid_address rep newp) =>
 (a, x, y, newp, b, T, ram) |
(let ir = (fetch rep (ram, address rep p)) in
 let m = (load_io rep (a, x, y, newp, ir, ram)) in
((FST m) => % invalid memory load %
(a, x, y, newp, b, T, ram) |
(let ldm = (SND m) in
 write_reg rep (a, x, y, newp, b, F, ir, ram, ldm))))))));

124
let WRITEIO = new_definition('WRITEIO',
"! (rep: rep_ty) (a:*wordn) (x:*wordn) (y:*wordn) (p:*wordn) (b:bool)
(stop:bool) (ram:*memory).  
WRITEIO rep (a, x, y, p, b, stop, ram) =
(stop => (a, x, y, p, b, stop, ram) |  
(let newp = (add rep (p, wordn rep 1)) in
(("valid_address rep newp) =>
(a, x, y, newp, b, T, ram) |  
(let ir = (fetch rep (ram, address rep p)) in
let value = load_r rep (a, x, y, newp, ir) in
let msfValue = (MSF rep ir) in
let addr = (address rep ir) in
((msfValue = (F,F)) => (a, x, y, newp, b, T, ram) |  
((msfValue = (F,T)) =>
(a, x, y, newp, b, F, (storeio rep (ram, addr, value))) |  
((msfValue = (T,F)) => (let t = (add rep (x, (pad rep addr))) in
((valid_address rep t) =>
(a, x, y, newp, b, F, (storeio rep(ram,(address rep t), value))) |  
(a, x, y, newp, b, T, ram)))) |  
(a, x, y, newp, b, T, ram))))))));

let WRITEM = new_definition('WRITEM',
"! (rep: rep_ty) (a:*wordn) (x:*wordn) (y:*wordn) (p:*wordn) (b:bool)
(stop:bool) (ram:*memory).  
WRITEM rep (a, x, y, p, b, stop, ram) =
(stop => (a, x, y, p, b, stop, ram) |  
(let newp = (add rep (p, wordn rep 1)) in
(("valid_address rep newp) =>
(a, x, y, newp, b, T, ram) |  
(let ir = (fetch rep (ram, address rep p)) in
let value = load_r rep (a, x, y, newp, it) in
let msfValue = (MSF rep ir) in
let addr = (address rep ir) in
((msfValue = (F,F)) => (a, x, y, newp, b, T, ram) |  
((msfValue = (F,T)) =>
(a, x, y, newp, b, F, (store rep (ram, addr, value))) |  
((msfValue = (T,F)) => (let t = (add rep (x, (pad rep addr))) in
((valid_address rep t) =>
(a, x, y, newp, b, F, (store rep(ram,(address rep t), value))) |  
(a, x, y, newp, b, T, ram)))) |  
(a, x, y, newp, b, T, ram))))))));
let NOOP_M = new_definition('NOOP_M',
  "! (rep:'rep_ty) (a:*wordn) (x:*wordn) (y:*wordn) (p:*wordn) (b:bool)
  (stop:bool) (ram:*memory) .
  NOOP_M rep (a, x, y, p, b, stop, ram) =
  (stop => (a, x, y, p, b, stop, ram)) |
  (let newp = (add rep (p, wordn rep 1)) in
   ("valid_address rep newp) =>
   (a, x, y, newp, b, T, ram))");;

let macro_state = "':((*wordn)#(*wordn)#(*wordn)#(*wordn)#bool#bool#(*memory))";;
let macro_env = "':(bool)'+;;

ABS_ENV takes a function of type (macro_state -> macro_state)
and creates a function of type (macro_state -> macro_env -> macro_state).
The purpose of this function is to make the functions defining the
instructions have the right type for use in the instruction list.

let ABS_ENV = new_definition
  ('ABS_ENV',
  "! (f:'macro_state->'macro_state) (x:'macro_state) (y:'macro_env) .
   ABS_ENV f x y = f x"
  );;

let macro_inst_list = new_definition
  ('macro_inst_list',
  "! (rep:'rep_ty').
  macro_inst_list rep =
  [ ((F,F,F,F),ABS_ENV (NOOP_M rep));
     ((F,F,F,T),ABS_ENV (SHR rep));
     ((F,F,F,F),ABS_ENV (SHRB rep));
     ((F,F,F,T),ABS_ENV (SHLB rep));
     ((F,F,T,F),ABS_ENV (SRL rep));
     ((F,F,T,F),ABS_ENV (SRLB rep));
     ((F,F,T,F),ABS_ENV (CMP rep));
     ((F,F,T,F),ABS_ENV (WRITE rep));
     ((F,F,T,T),ABS_ENV (WRITEIO rep));
     ((F,T,F,F),ABS_ENV (NEG rep));
   ]
  );;
let Opcode = new_definition ('Opcode', "! (rep:'rep_ty) (a:,wordn) (x:*wordn) (y:*wordn) (p:*wordn) (b:bool) (stop:bool) (ram:,memory) (reset:bool) . Opcode rep (a, x, y, p, b, stop, ram) (reset = (FST (SND (decode rep ((opcode rep (fetch rep (ram, address rep p))), b)))) ");

Opc_Val will be used to instantiate key in mk_macro.ml

%-------------------------------------------------------------

let Opc_Val = new_definition ('Opc_Val', "! (x:bt5) . Opc_Val x = (bt5_val x)"");

close_theory();
% new_inst_aux updates "update_reg" usage to include new "b" parameter

load_parent 'regs_def';
load_parent 'aux_thms';
load_parent 'macro_def';

let REG_LIST_LENGTH = new_definition
('REG_LIST_LENGTH',
 "REG_LIST_LENGTH (rep:'rep_ty) =
  ! (l:(,wordm) list) . (LENGTH l = p_reg)");;

%-------------------------------------------------------
Prove some facts about the independence of register updates
-------------------------------------------------------------%

let EL_SET_EL_TAC =
  REPEAT GEN_TAC
  THEN REWRITE_TAC[A:X:Y:P]
  THEN CONV_TAC (TOP_DEPTH_CONV num_CONV)
  THEN REWRITE_TAC[LENGTH_CONS]
  THEN DISCH_TAC
  THEN POP_ASSUM(
      \p. MAP_EVERY ASSUME_TAC(
        rev (CONJUNCTS \p))
  )
  THENL
    [ALL_TAC
      ;
      POP_ASSUM(
        \p. DISJ_CASES_TAC \p)
    ]
  THEN POP_ASSUM(
      \p. ASSUME_TAC (REWRITE_RULE[theCase] \p))
  THEN REWRITE_TAC [theCase]
  THEN POP_ASSUM(
      \p. CHOOSE_THEN CHOOSE_TAC \p)
  THEN POP_ASSUM(
      \p. MAP_EVERY ASSUME_TAC (rev(CONJUNCTS \p))
  )
  THEN POP_ASSUM(
      \p. CHOOSE_THEN CHOOSE_TAC \p)
  THEN POP_ASSUM(
      \p. MAP_EVERY ASSUME_TAC (rev(CONJUNCTS \p))
  )
  THEN ASM_REWRITE_TAC []
  THEN CONV_TAC (TOP_DEPTH_CONV num_CONV)
  THEN REWRITE_TAC [EL;SET_EL;HD;TL];;

%-----------------------------------------------------------------
Run time: 76.5s
Intermediate theorems generated: 4412
-----------------------------------------------------------------

let INDEP_A_UPDATE = prove_thm('INDEP_A_UPDATE',..
let INDEP_X_UPDATE = prove_thm('INDEP_X_UPDATE',

  "! (n:num) (l:*wordn list) (z:*wordn).
  ( ((n = a_reg) \/ (n = y_reg) \/ (n = p_reg)) \/
    (LENGTH 1 = p_reg))
  ==> ((EL n (SET_EL a_reg 1 z) ) = EL n 1)",
  EL_SET_EL_TAC );;

let INDEP_Y_UPDATE = prove_thm('INDEP_Y_UPDATE',

  "! (n:num) (l:*wordn list) (z:_wordn).
  ( ((n = a_reg) \/ (n = x_reg) \/ (n = p_reg)) \/
    (LENGTH 1 = p_reg))
  ==> ((EL n (SET_EL y_reg 1 z) ) = EL n 1)",
  EL_SET_EL_TAC );;

let INDEP_P_UPDATE = prove_thm('INDEP_P_UPDATE',

  "! (n:num) (l:*wordn list) (z:*wordn).
  ( ((n = a_reg) \/ (n = x_reg) \/ (n = y_reg)) \/
    (LENGTH 1 = p_reg))
  ==> ((EL n (SET_EL p_reg 1 z) ) = EL n 1)",
  EL_SET_EL_TAC );;

let LISTa_CORRECT = prove_thm('LISTa_CORRECT',

  "! (l:*wordn) list) b t. (LENGTH 1 = p_reg) =>
  (EL a_reg (update_reg l (F,T,T) (b (t:num))
  (add (rep:'rep ty) (EL p_reg l, wordn rep 1)) =
  EL a_reg l )",
  REPEAT GEN_TAC
  THEN DISCH_TAC
  THEN REWRITE_TAC [update_reg;PAIR_EQ]
  THEN
  IMP_RES_TAC (REWRITE_RULE []
    (SPECL ['"a_reg"":"1:*wordn list";
    "(add (rep:'rep ty)(EL p_reg l,wordn rep 1))"])

%-----------------------------------------------------------

LIST?_CORRECT

Run time: 49.9s
Intermediate theorems generated: 1467
%-----------------------------------------------------------
let LISTx_CORRECT = prove_thm('LISTx_CORRECT',
    "(!l:(\*wordn) list) b t. (LENGTH l = p_reg) ==> 
    (EL x_reg (update_reg l (F,T,T) (b (t:num)) 
    (add (rep:"rep_ty) (EL p_reg l, wordn rep 1))) = 
    EL x_reg l)."
);

let LISTy_CORRECT = prove_thm('LISTy_CORRECT',
    "(!l:(\*wordn) list) b t. (LENGTH 1 = p_reg) ==> 
    (EL y_reg (update_reg l (F,T,T) (b (t:num)) 
    (add (rep:"rep_ty) (EL p_reg l, wordn rep 1))) = 
    EL y_reg l)."
);

map (delete_cache o fst) (cached_theories());;

Use this to generate goals for correct instantiation (implementation) proofs.

%-----------------------------------------------------------
% This redefines the one in mk_mac_.
%-----------------------------------------------------------

let MK_INST_CORRECT_GOAL n =
  let inst = term_list_el n 
    (snd(dest_eq( 
        snd(dest_forall(concl macro_inst_list)))) in 
  "(!rep:"rep_ty) (regs:time->\*wordn list) (m ins din dout:time->\*wordn) 
  (ram:time->\*memory) (b stop ovl:time->bool) (mar:time->\*address) 
  (REG_LIST_LENGTH rep \/

130
DECODE_M_CORRECTLY_IMP rep) =>
(Macro_Int_IMPLIMP rep
(
(
reg t,m t,ins t,din t,dout t,ram t,b t,stop t,ovl t,
res t,mpc t)
(
reset_e t) "inst)";)

%--------------------------------------------------------------
definitions for symbolic execution
%--------------------------------------------------------------%

let int_to_term = ((C o curry) mk_const "::num") o string_of_int and
term_to_int = (int_of_string o fst o dest_const);;

let sum_to_term x y = int_to_term (x+y);;

let sumTerm x y =
  mk_comb (mk_comb (mk_const ('+',
    mk_type('fun',[mk_type('num',[])];
    mk_type('fun',[ mk_type('num',[]);
      mk_type('num',[])]))),
  mk_type('fun',[ mk_type('num',[])])
  mk_const( (string_of_int x), mk_type('num',[]))),
  mk_type('fun',[ mk_type('num',[])])
  mk_const( (string_of_int y), mk_type('num',[])));

let t_plus_term y =
  mk_comb (mk_comb (mk_comb ('+',
    mk_type('fun',[mk_type('num',[])];
    mk_type('fun',[ mk_type('num',[]);
      mk_type('num',[])]))),
  mk_type('fun',[ mk_type('num',[])])
  mk_var( 't', mk_type('num',[])))
  mk_type('fun',[ mk_type('num',[])])
  mk_type('num',[])));

let sumTHM x y =
  REWRITE_RULE 
  [REWRITE_RULE [ADD_CLAUSES;
    SYM_RULE((TOP_DEPTH_CONV num_CONV) (sum_to_term x y ) )]
  ((TOP_DEPTH_CONV num_CONV) (sumTerm x y ))]
((SPECL ["t"; int_to_term x; int_to_term y] (SYM_RULE ADD_ASSOC)));

%--------------------------------------------------------------

let T_DIFF_TAC x y =
  REWRITE_TAC [SPECL ["t";x;y] (SYM_RULE ADD_ASSOC)]
  THEN CONV_TAC (TOP_DEPTH_CONV num_CONV)
  THEN REWRITE_TAC [ADD_CLAUSES];;

set_goal("
(t+3)+4 = t+7"
);

e( T_DIFF_TAC "3" "4" );;

%--------------------------------------------------------------
let PLUS_ONE_TAC n = 
  REWRITE_TAC [(SYM_RULE ADDI);(NUM_CONV n);ADD_CLAUSES];;

let T2 = prove_thm ('T2', "!t. (t + 1) + 1 = t + 2", PLUS_ONE_TAC "2");;
let T3 = prove_thm ('T3', "!t. (t + 2) + 1 = t + 3", PLUS_ONE_TAC "3");;
let T4 = prove_thm ('T4', "!t. (t + 3) + 1 = t + 4", PLUS_ONE_TAC "4");;
let T5 = prove_thm ('T5', "!t. (t + 4) + 1 = t + 5", PLUS_ONE_TAC "5");;
let T6 = prove_thm ('T6', "!t. (t + 5) + 1 = t + 6", PLUS_ONE_TAC "6");;
let T7 = prove_thm ('T7', "!t. (t + 6) + 1 = t + 7", PLUS_ONE_TAC "7");;
let T8 = prove_thm ('T8', "!t. (t + 7) + 1 = t + 8", PLUS_ONE_TAC "8");;
let T9 = prove_thm ('T9', "!t. (t + 8) + 1 = t + 9", PLUS_ONE_TAC "9");;
let T10 = prove_thm ('T10', "!t. (t + 9) + 1 = t + 10", PLUS_ONE_TAC "10");;
let T11 = prove_thm ('T11', "!t. (t + 10) + 1 = t + 11", PLUS_ONE_TAC "11");;
let T12 = prove_thm ('T12', "!t. (t + 11) + 1 = t + 12", PLUS_ONE_TAC "12");;
let T13 = prove_thm ('T13', "!t. (t + 12) + 1 = t + 13", PLUS_ONE_TAC "13");;
let T14 = prove_thm ('T14', "!t. (t + 13) + 1 = t + 14", PLUS_ONE_TAC "14");;
let T15 = prove_thm ('T15', "!t. (t + 14) + 1 = t + 15", PLUS_ONE_TAC "15");;
let T16 = prove_thm ('T16', "!t. (t + 15) + 1 = t + 16", PLUS_ONE_TAC "16");;
let T17 = prove_thm ('T17', "!t. (t + 16) + 1 = t + 17", PLUS_ONE_TAC "17");;
let T18 = prove_thm ('T18', "!t. (t + 17) + 1 = t + 18", PLUS_ONE_TAC "18");;

let T_THMS = [T2; T3; T4; T5; T6; T7; T8; T9; T10; T11; T12; T13; T14; T15; T16; T17; T18];;

%---

Define the relationship between selectors op and address and the constructor join
%---

let op_join_op = mk_thm([], "!fet ad.
  (opcode rep:'rep_ty)
  (join rep
   (opcode rep(fet),
    address rep(ad))) = opcode rep(fet)");;

let address_join_address = mk_thm([], "!fet ad.
  (address rep:'rep_ty)
  (join rep
   (opcode rep(fet),
    address rep(ad))) = address rep(ad)");;

let DSF_join_op = mk_thm([], "!fet ad.
  (DSF (rep:'rep_ty)
(join rep
  (opcode rep(fet),
   address rep(ad) )) = DSF rep(fet);;

let RSF_join_op = mk_thm([], "!fet ad.
  (RSF (rep:'rep_ty)
    (join rep
     (opcode rep(fet),
      address rep(ad) ))) = HSF rep(fet)";;

let address_pad_address = mk_thm([], "!w.
  (address (rep:'rep_ty) (pad rep (address rep(w)))) =
   address rep( w )";;

let FSF_join_op = mk_thm([], "!fet ad.
  (FSF (rep:'rep_ty)
    (join rep
     (opcode rep(fet),
      address rep(ad) ))) = FSF rep(fet)";;

%--------------------------------------------------------------------------
NORMAL_SYMB_EXEC takes as arguments a microinstruction to expand
and one of the "T" theorems from above

should append _TAC to the name!

--------------------------------------------------------------------------%

let NORMAL_SYMB_EXEC n T =
  IMP_RES_TAC (el n Micro_Int_Inst_list)
  THEN ASSUM_LIST (asl. POP_ASSUM(thm. POP_ASSUM(thm1.
    % note that the thm1 are not used %
    MAP_EVERY ASSUME_TAC (CONJUNCTS (REWRITE_RULE
      ([PAIR_EQ;T;op_join_op;address_join_address;
        DSF_join_op;RSF_join_op;address_pad_address]
     @ (subtract asl[(el I asl)]) (el 1 asl)) ))
    THEN NORMAL_POP_ASSUM_TAC ;;

%--------------------------------------------------------------------------
NEXT_SYMB_EXEC_TAC determines what the next microinstruction
expansion should be based on the mpc (on top of assumption stack).
It then invokes NORMAL_SYMB_EXEC passing one of the T_THMS.

(term_to_int(bt_val_func(snd(dest_eq(snd(dest_thm( (el I asl)))))))+1) t);;
%--------------------------------------------------------------------------%
let mpc_from_thm thm =
  (term_to_int(bt_val_func(snd(dest_thm(thm)))));

let NEXT_SYMB_EXEC_TAC theTime =
  let t = (el theTime T_THMS) in
  ASSUM_LIST(asl. NORMAL_SYMB_EXEC (mpc_from_thm (el 1 asl)+1) t);;

% CASES_NEXT_SYMB_EXEC_TAC may be outdated %

let CASES_NEXT_SYMB_EXEC_TAC theTime theCond =
  let t = (el theTime T_THMS) in
  ASSUM_LIST(asl.
    IMP_RES_TAC (el (mpc_from_thm (el 1 asl)+1) Micro_Int Inst_list))
  THEN ASM_CASES_TAC theCond
  THEN ASSUM_LIST(kasl.
    IMP_RES_TAC (el (mpc_from_thm (el I asl)+l) Micro_Int Inst_list))
  THEN NORMAL_POP_ASSUM_TAC ;;

The following definitions help remove unneeded theorems from the assertion list. After using NORMAL_SYMB_EXEC, there are many theorems from the previous step that can be eliminated.

The tactic DELETE_USTEP_TAC expects a number argument and removes all theorems from the assumption list corresponding to that time.

let is_at_time_of utime tok =
  if( is_eq tok)
    then (let 1 = lhs(tok) in % mar(t+1) %
      if( is_comb 1 )
        then (let r = rand(1) in % (t+1) %
          if( is_comb r )
            then (let op = rator(r) in
              if( op = "+ t")
                then (if( rand(r) = int_to_term(utime) )
                then
                  if( rator(1) = "mpc") then

(print_ibegin 0; print_term tok; print_end();
  print_newline(); true)
else true
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else false
)
else pair_el (n-1) (snd(dest_pair(p)));

The following tactic converts an assumption like
[ "mpc(t + 6) = bt7_ival(6 + (bt5_val(F,F,F,F,F))"
]
to:
[ "mpc(t + 6) = F,F,F,T,F,T"
]
It is a modified version of NORMAL_POP_ASSUM_TAC
(who picked that tactic name anyway? :-)

let JMPOPC_POP_ASSUM_TAC =
  POP_ASSUM (\thm. ASSUME_TAC (CONV_RULE (ONCE_DEPTH_CONV bt7_ival_CONV) (CONV_RULE DEC_ADD_CONV (PURE_ONCE_REWRITE_RULE [ADD_CLAUSES] (CONV_RULE (DEC_ADD_CONV broken for "0 + 1" % REWRITE_RULE [add_btT] thm))))));

map (delete_cache o fst) (cached_theories());

let FETCH_INST_TAC n = % set up everything for all proofs? %
  let thm = el (n+1) macro_defn_list in (
    let inst_lemma = EXPAND_LET_RULE thm
    and inst = term_list_el n
    (snd(dest_eq(_snd(dest_forall(concl macro_inst_list)))))
    (REPEAT GEN_TAC
     THEN STRIP_TAC
     THEN SUBST_TAC [SPEC inst Macro_Int_IMPL_IMPL_LEMMA]
     THEN ASM_REWRITE_TAC [inst_lemma;ABS_ENV])
    (REWRITE_TAC [KEG_LIST_LENGTH] (el 5 asl))
  )

let JMPOP_C_POP_ASSUM_TAC =
  POP_ASSUM (\thm. ASSUME_TAC (CONV_RULE (ONCE_DEPTH_CONV bt7_ival_CONV) (CONV_RULE DEC_ADD_CONV (PURE_ONCE_REWRITE_RULE [ADD_CLAUSES] (CONV_RULE (DEC_ADD_CONV broken for "0 + 1" % REWRITE_RULE [add_btT] thm))))));

map (delete_cache o fst) (cached_theories());

let FETCH_INST_TAC n = % set up everything for all proofs? %
  let thm = el (n+1) macro_defn_list in (REPEAT GEN_TAC
  
   REWRITE_RULE [KEG_LIST_LENGTH] (el 5 asl))

THEN ASSUM_LIST (asl. ASSUME_TAC (REWRITE_RULE[el 2 asl];PAIR_EQ)
(EXPAND_LEM_RULE (SPEC
  ["fetch (rep:'rep_ty) (ram (t:num), address rep (EL p_reg (reg (t:num)))):*wordn";
  "(b (t:num)):$bool"]
  (PURE_REWRITE_RULE[DECODE_M_CORRECTLY_IMP] (el 4 asl))))
THEN ASSUM_LIST (asl. ASSUME_TAC (REWRITE_RULE[el 3 asl];PAIR_EQ) (SPEC
  (fst(dest_eq(snd(dest_thm(el 3 asl)))) MacroLevelCycles))
THEN ASM_REWRITE_TAC[]
% take care of stop case %
THEN ASM_CASES_TAC "(stop (t:num)):bool"
THEN ASSUM_LIST (asl. REWRITE_TAC [el 1 asl])
THENL [ % subgoal 1 (stop t) %
  ASSUM_LIST (asl. IMP_RES_TAC
    (SPEC [snd(dest_eq(snd(dest_thm(el 2 asl))))];"t:num") stop_thm))
THEN ASSUM_LIST (asl. MAP_EVERY ASSUME_TAC
  (CONJUNCTS (REWRITE_RULE [PAIR_EQ] (el 1 asl))))
THEN ASM_REWRITE_TAC[PAIR_EQ]
; % subgoal 2 - stop t %
NORMAL_SYMB_EXEC 1 T2 % T2 here is a placeholder %
THEN NORMAL_SYMB_EXEC 2 T2
THEN COND_CASES_TAC
THENL [ % subgoal 2.1 "valid_address %
  NORMAL_SYMB_EXEC 3 T3
  % The processor is now stopped due to an addressing exception %
  % specialize and rewrite stop_thm show nothing will change %
  THEN ASSUM_LIST (asl. ASSUME_TAC REWRITE_RULE
    [(el 5 asl);(el 43 asl);(el 1 asl)] (SPECL [(int_to_term
      ((term_to_int (snd(dest_eq(snd(dest_thm.
        (REWRITE_RULE [PAIR_EQ] (el 39 asl))))))))-3));
      "+(t+3):num"] stop_thm)) ))
THEN ASSUM_LIST (asl. POP_ASSUM(thm.
  (MAP_EVERY ASSUME_TAC (CONJUNCTS (REWRITE_RULE
    [(PAIR_EQ); (sumTHM 3
      ((term_to_int(snd(dest_eq(snd(dest_thm
        (el 40 asl)))))))-3))
    ) 0 (subtract asl[(el 1 asl)]))) (el 1 asl)) ))))
THEN ASM_REWRITE_TAC [PAIR_EQ]
THEN REWRITE_TAC [update_reg; PAIR_EQ;EL_SET_EL]
; % subgoal 2.2 valid_address %
POP_ASSUM(\thm. ASSUME_TAC (REWRITE_RULE [] thm))
THEN NORMAL_SYMB_EXEC 3 T3
THEN DELETE_USTEP_TAC 1 THEN DELETE_USTEP_TAC 2
]];:
let INDEP_REG_TAC aReg INDEP_THM =

ASSUM_LIST(asl. REWRITE_TAC
[(REWRITE_RULE[(SPEC "(update_reg(reg (t:num))(F,T,T) \
(add rep(EL p_reg(reg t),wordn (rep:"rep_ty) 1))")
(REWRITE_RULE[REG_LIST_LENGTH] (last asl ))]]
(SPEC [aReg; "(update_reg(reg (t:num))(F,T,T) \
(add rep(EL p_reg(reg t),wordn (rep:"rep_ty) 1))")" ] INDEP_THM))];;

% |- EL 1 = EL x_reg %
let ELX = AP_TERM "EL:num->(*wordn)list->*wordn" (SYM X);;

% |- SET_EL 1 = SET_EL x_reg %
let SET_ELX = AP_TERM "SET_EL:num->(*vordn)list->(*wordn->(*wordn)list)"
(SYM X);;

let THREE_TUPLE_CASES_ASSOC = prove_thm('THREE_TUPLE_CASES_ASSOC',
"!b. 
((b = T,T,T) \ (b = F,T,T)) \ (b = T,F,T) \ (b = F,F,T)) \ 
((b = T,T,F) \ (b = F,T,F)) \ 
(b = T,F,F) \ 
(b = F,F,F) 
= ( (b = F,F,F) \ 
(b = F,F,T) \ 
(b = F,T,F) \ 
(b = F,T,T) \ 
(b = T,F,F) \ 
(b = T,F,T) \ 
(b = T,T,F) \ 
(b = T,T,F) 
(b = T,T,T) )",

GEN_TAC
THEN ASM_CASES_TAC "(b = F,F,F)"
THENL[ ALL_TAC; ASM_CASES_TAC "(b = F,F,T)"
THENL[ ALL_TAC; ASM_CASES_TAC "(b = F,T,F)"
THENL[ ALL_TAC; ASM_CASES_TAC "(b = F,T,T)"
THENL[ ALL_TAC; ASM_CASES_TAC "(b = T,F,F)"
THENL[ ALL_TAC; ASM_CASES_TAC "(b = T,F,T)"
THENL[ ALL_TAC; ASM_CASES_TAC "(b = T,T,F)"
THENL[ ALL_TAC; ASM_CASES_TAC "(b = T,T,T)"
]]]]))]
THEN ASM_REWRITE_TAC [OR_CLAUSES:PAIR_EQ]
let THREE_TUPLE_VALUE_ASSOC_LEMMA = prove_thm('THREE_TUPLE_VALUE_ASSOC_LEMMA',
"!b. (b = F,F,F) \/
    (b = F,F,T) \/
    (b = F,T,F) \/
    (b = F,T,T) \/
    (b = T,F,F) \/
    (b = T,F,T) \/
    (b = T,T,F) \/
    (b = T,T,T)",(GEN_TAC
    THEN SUBST_TAC [SYM (SPEC "b" THREE_TUPLE_CASES_ASSOC)]
    THEN REWRITE_TAC[(SPEC "b" THREE_TUPLE_VALUE_LEMMA)];;

let THREE_TUPLE_IMPI = prove_thm('THREE_TUPLE_IMPI',
"!b. ((b = F,F,F) \/
     (b = F,F,T) \/
     (b = F,T,F) ) =>
     -((b = F,T,T) \/
     (b = T,F,F) \/
     (b = T,F,T) \/
     (b = T,T,F) \/
     (b = T,T,T))", (GEN_TAC
    THEN DISCH_TAC
    THEN POP_ASSUM(
thm. DISJ_CASES_TAC thm)
    THEN (POP_ASSUM(
thm. DISJ_CASES_TAC thm) ORELSE ALL_TAC)
    THEN ASM_REWRITE_TAC [PAIR_EQ]);;

let RSF_CASES = SPEC
"(RSF (rep:-rep_ty)(fetch rep(ram t,address rep(EL p_reg(reg t)))))
   TWO_TUPLE_VALUE_LEMMA;;

let DSF_CASES = SPEC
"(DSF (rep:-rep_ty)(fetch rep(ram t,address rep(EL p_reg(reg t))))
   THREE_TUPLE_VALUE_ASSOC_LEMMA;;

let AXY_DSF_CASES =
""((DSF (rep:-rep_ty)(fetch rep(ram t,address rep(EL p_reg(reg t)))) = F,F,F)
\/
    (DSF rep(fetch rep(ram t,address rep(EL p_reg(reg t)))) = F,F,T)
\/
    (DSF rep(fetch rep(ram t,address rep(EL p_reg(reg t)))) = F,T,F)";;

let AXY_IMP1 = (SPEC
"(DSF (rep:-rep_ty)(fetch rep(ram t,address rep(EL p_reg(reg t))))
   THREE_TUPLE_IMP1);;
let RSF_CASES_TAC =
  DISJ_CASES_TAC RSF_CASES % cond on RSF - 4 subgoals proved%
  THEN POP_ASSUM(
    \thm. DISJ_CASES_TAC thm)
  THEN % rewrite (reg t+10) with the conditions and asl%
    ASSUM_LIST(asl. let regsVal = (el 14 asl) in ASSUME_TAC(
      REWRITE_RULE [PAIR_EQ; bt3_val; (SYM A);(SYM Y);(SYM P);ELX;SET_ELX]
      (ONCE_REWRITE_RULE[update_reg]
       (REWRITE_RULE ( subtract asl[regsVal] @ 
         [bt2_val;bt3_val; (SYM A);(SYM Y);(SYM P)]) regsVal)))
    THEN ASM_REWRITE_TAC [PAIR_EQ;EL_SET_EL];;

let ELP_SET_ELP = TAC_PROOF (([], "!(newVal:*wordn) b.
  (EL p_reg (update_reg (reg (t:num)) (F,T,T) b newVal)) = newVal"),
  REPEAT 0EN_TAC
  THEN REWRITE_TAC [update_reg;bt3_val;(SYM P);EL_SET_EL;PAIR_EQ] );;

let EL_COND_THM = TAC_PROOF (([], "!(regs:*wordn list) sel.
  (EL ( sel = F,F ) => a_reg | 
   (sel = F,T) => x_reg | 
   (sel = T,F) => y_reg | 
   p_reg ) regs ) =  
  ((sel = F,F) => EL a_reg regs | 
  (sel = F,T) => EL x_reg regs | 
  (sel = T,F) => EL y_reg regs | 
  EL p_reg regs )")

  GEN_TAC THEN GEN_TAC
  THEN COND_CASES_TAC
  THEN REWRITE_TAC []
  THEN COND_CASES_TAC
  THEN REWRITE_TAC []
  THEN COND_CASES_TAC
  THEN REWRITE_TAC []
  );;

let SPECI_EL_COND_THM =
  SPECL [("(update_reg((reg (t:num)):*wordn list)(F,T,T)(b t)
    (add (rep:^rep_ty)(EL p_reg(reg t),wordn rep 1)))";
  "(RSF (rep:^rep_ty)(fetch rep(ram (t:num),
    address rep(EL p_reg(reg t))))"]
  EL_COND_THM;:

let bt2_reg_def = REWRITE_RULE [(SYM A);(SYM X);(SYM Y);(SYM P)] bt2_val_def;;

let INDEP_A_UPDATE1 = prove_thm('INDEP_A_UPDATE1',

"!(l:*wordn list) (n:num) (z:*wordn).
( ((n = x_reg) \/ (n = y_reg) \/ (n = p_reg)) \/
(LENGTH l = p_reg))
=> ((EL n (SET_EL a_reg l z) ) = EL n l)".
EL_SET_EL_TAC ;;

let INDEP_X_UPDATEI = prove_thm('INDEP_X_UPDATEI',
"!(l:*wordn list) (n:num) (z:*wordn).
( ((n = a_reg) \/ (n = y_reg) \/ (n = p_reg)) \/
(LENGTH l = p_reg))
=> ((EL n (SET_EL x_reg l z) ) = EL n l)".
EL_SET_EL_TAC ;;

let INDEP_Y_UPDATEI = prove_thm('INDEP_Y_UPDATEI',
"!(l:*wordn list) (n:num) (z:*wordn).
( ((n = a_reg) \/ (n = x_reg) \/ (n = p_reg)) \/
(LENGTH l = p_reg))
=> ((EL n (SET_EL y_reg l z) ) = EL n l)".
EL_SET_EL_TAC ;;

let INDEPENDENCE_TAC UPDATE_THM = % 325.6 %
ASSUM_LIST(asl. ASSUME_TAC(
(REWITE_RULE (( SPEC "(update_reg(reg (t:num))(F,T,T)(b t)
(add rep(EL p_reg(reg t),wordn (rep:"rep_ty") 1))))")
(REWRITE_RULE [REG_LIST_LENGTH] (last asl)) )]
(SPEC [(update_reg(reg (t:num))(F,T,T)(b t)
(add rep(EL p_reg(reg t),wordn (rep:"rep_ty") 1))))] UPDATE_THM )))
THEN POP_ASSUM(/\thm. REWRITE_TAC [ELP_SET_ELP;
(REWRITE_RULE(SPEC "a_reg" thm ));
(REWRITE_RULE(SPEC "x_reg" thm ));
(REWRITE_RULE(SPEC "y_reg" thm ));
(REWRITE_RULE(SPEC "p_reg" thm )) ) ));

let EXPAND_REG_TAC = % 235.0s %
ASSUM_LIST(asl. let regsVal = (el 13 asl) in ASSUME_TAC(
REWRITE_RULE [PAIR_EQ;bt3_val;(SYM A);(SYM Y);(SYM P);ELX;SET_ELX;
bt2_reg_def;SPEC1_EL_COND_THM;ELP_SET_ELP]
(ONCE_REWRITE_RULE[update_reg]
(REWRITE_RULE ( (subtract asl[regsVal]) @
[bt2_val;bt3_val;(SYM A);(SYM Y);(SYM P)]) regsVal ))) );

let EXPAND_B_TAC = % 235.0s %
ASSUM_LIST(asl. let bVal = (el 8 asl) in ASSUME_TAC(
REWRITE_RULE [PAIR_EQ;bt3_val;(SYM A);(SYM Y);(SYM P);ELX;SET_ELX;
bt2_reg_def;EL_COND_THM;ELP_SET_ELP]
let EXPAND_COND_TAC thmNum =
  ASSUM_LIST(asl. let thm = (el thmNum asl) in ASSUME_TAC(
    REWRITE_RULE [PAIR_EQ;bt3_val;(SYM A);(SYM Y);(SYM P);ELX;SET_ELX;
      bt2_reg_def;EL_COND_THM;ELP_SET_ELP]
    (REWRITE_RULE ( (subtract asl[thm])
      (bt2_val;bt3_val;(SYM A);(SYM Y);(SYM P)) thm )) );;

let FETCH_OPERAND_CASES_TAC =
  NORMAL_SYMB_EXEC 4 T4 THEN DELETE_USTEP_TAC 3
  THEN NORMAL_SYMB_EXEC 5 T5 THEN DELETE_USTEP_TAC 4
  THEN REWRITE_TAC [load_m_expanded; write_reg_expanded; load_r_expanded;
    write_reg_expanded]

  % construct MSF cases %
  THEN ASSUM_LIST(asl. ASSUME_TAC ( SPEC (snd(dest_comb(snd(dest_comb(snd(
    dest_comb(rhs(snd(dest_th_J((el I asl))))))))))
      TWO_TUPLE_VALUE_LEMMA))
  THEN POP_ASSUM(
    thm. DISJ_CASES_TAC thm)
  THEN POP_ASSUM(
    thm. DISJ_CASES_TAC thm);;
let FIND_ASSUME f asl = hd(filter(f o concl) asl);

let MSF_CASE_MPC_REWRITE_TAC = 
  ASSUM_LIST(asl. ASSUME_TAC(REWRITE_RULE[(el 1 asl):bt2_val] 
  (el 2 asl)))
  THEN POP_ASSUM(\thm. ASSUME_TAC( 
   CONV_RULE(ONCE_DEPTH_CONV bt7_iival_CONV) ( 
   CONV_RULE DEC_ADD_CONV ( 
     % DEC_ADD_CONV broken for "0 + 1" % 
     PURE_ONCE_REWRITE_RULE[ADD_CLAUSES] (thm))))));

let MSF_FT_FF_FETCH_TAC = 
  ASSUM_LIST(\asl. REWRITE_TAC[(el I asl);PAIR_EQ] ) % 2567.8s % 
  THEN MSF_CASE_MPC_REWRITE_TAC 
  THEN NEXT_SYMB_EXEC_TAC 6 THEN DELETE_USTEP_TAC 5 
  THEN NEXT_SYMB_EXEC_TAC 7 THEN DELETE_USTEP_TAC 6 
  THEN NEXT_SYMB_EXEC_TAC 8 THEN DELETE_USTEP_TAC 7 
  THEN NEXT_SYMB_EXEC_TAC 9 THEN DELETE_USTEP_TAC 8 
  THEN NEXT_SYMB_EXEC_TAC 10 THEN DELETE_USTEP_TAC 9 
  THEN NEXT_SYMB_EXEC_TAC 11 THEN DELETE_USTEP_TAC 10 
  THEN NEXT_SYMB_EXEC_TAC 12 
  THEN JMPPOPC_POP_ASSUM_TAC THEN DELETE_USTEP_TAC 11;;

let SYMB_EXEC_ASSUME_TAC mpcAsm theTimeThm = 
  ASSUM_LIST(\asl. 
   IMP_RES_TAC (el (mpc_from_thm (el mpcAsm asl)+1) Micro_Int_Inst_list)) 
  THEN POP_ASSUM(\thm. POP_ASSUM(\thml. ASSUM_LIST(kasl. ASSUME_TAC( 
   (REWRITE_RULE([theTimeThm] @ asl) thm) ))));

let SYMB_EXEC_ASSUME_TACI mpcAsm theTimeThm = 
  ASSUM_LIST(kasl. 
   IMP_RES_TAC (el (mpc_from_thm (el mpcAsm asl)+1) Micro_Int_Inst_list)) 
  THEN POP_ASSUM(\thm. POP_ASSUM(\thml. ASSUM_LIST(kasl. MAP_EVERY ASSUME_TAC (CDNJUNCTS(REWRITE_RULE([PAIR_EQ;theTimeThm;DSF_join_op;op_join_op;address_join_address; 
   address_pad_address] @ asl) thm)) @ asl) thm) )

% The processor is now stopped due to an addressing exception %
% specialize and rewrite stop_thm show nothing will change %
let EXTEND_STOP_TAC when M_I_thm = 
  ASSUM_LIST(\asl. 
   let curTime = (term_to_int (rand(rand(fst( dest_eq(snd(dest_thm(el 1 asl)))))))) in 
   let endTime = 
     (term_to_int (snd(dest_eq(snd(dest_thm (el when asl))))) in 
   ASSUME_TAC( REWRITE_RULE [(el 1 asl); (el 5 asl); (el M_I_thm asl); 
     (sumTHM curTime (endTime-curTime))] )

143
(SPECL [(int_to_term (endTime - curTime )); (t_plus_term curTime)];
stop_thm)
)

THEN POP_ASSUM( \thm. REWRITE_TAC [(REWRITE_RULE [PAIR_EQ] thm)] )

THEN ASM_REWRITE_TAC []

THEN REWRITE_TAC [ELP_SET_ELP];;

let GOOD_DEST_TAC =
ASSUM_LIST ( \asl. DISJ_CASES_TAC (el 14 asl) )
THEN

[ EXPAND_REG_TAC
THEN EXPAND_B_TAC
THEN ASM_REWRITE_TAC [PAIR_EQ;EL_SET_EL;DSF_join_op;op_join_op);
   address_join_address; address_pad_address]

THEN INDEPENDENCE_TAC INDEP_A_UPDATE1
;
POP_ASSUM( \thm. DISJ_CASES_TAC thm)
THEN EXPAND_REG_TAC
THEN EXPAND_B_TAC
THEN ASM_REWRITE_TAC [PAIR_EQ;EL_SET_EL;DSF_join_op;op_join_op);
   address_join_address; address_pad_address]

THENL

[ INDEPENDENCE_TAC INDEP_X_UPDATE1;
   INDEPENDENCE_TAC INDEP_Y_UPDATE1
  ];
];;

let MSF_TF.FETCH_TAC =
ASSUM_LIST ( \asl. REWRITE_TAC[(el 1 asl);PAIR_EQ ] ) % 2567.8s %

THEN NSF_CASE_MPC_REWRITE_TAC
THEN NEXT_SYMB_EXEC_TAC 6 THEN DELETE_USTEP_TAC 5
THEN NEXT_SYMB_EXEC_TAC 7 THEN DELETE_USTEP_TAC 6
THEN NEXT_SYMB_EXEC_TAC 8 THEN DELETE_USTEP_TAC 7
THEN SYMB_EXEC_ASSUM_TAC I T9
% case split based on valid address %

THEN ASSUM_LIST ( \asl. ASM_CASES_TAC ( (fst(dest_cond
   (pair_el 9 (snd(dest_eq(snd(dest_thm(el 1 asl)))))))) )
)
THENL

[ %----------------------- `valid address------------- %
   POP_ASSUM( \theCase. POP_ASSUM( \thm. MAP_EVERY_ASSUME_TAC ( [
      theCase] @ (CONJUNCTS
   (REWRITE_RULE [PAIR_EQ; theCase] thm)))))

THEN DELETE_USTEP_TAC 8
THEN ASSUM_LIST ( \asl. ASSUME_TAC % (el 13 asl) is valid_address ... %
   (REWRITE_RULE [PAIR_EQ;update_reg] (el 13 asl)) )

THEN ASM_REWRITE_TAC [PAIR_EQ]
% The processor is now stopped due to an addressing exception %
% specialize and rewrite stop_thm show nothing will change %
THEN ASSUM_LIST(\asl.
    let MLC tok = (rator(lhs tok)) = "MacroLevelCycles" ? false in
    let endTimeThm = (FIND_ASSUM MLC asl) in
    let curTime = (term_to_int
        (rand(rand(fst(dest_eq(snd(dest_thm(el 2 asl))))))))) in
    let endTime =
        (term_to_int
            (snd(dest_eq(snd(dest_thm(endTimeThm))))))) in
    ASSUME_TAC(REWRITE_RULE (aslO
        (sumTHM curTime (endTime-cumTime))))

THEN POP_ASSUM(\thm. REWRITE_TAC [(REWRITE_RULE [PAIR_EQ] thm)])
THEN ASM_REWRITE_TAC []
THEN REWRITE_TAC [ELP_SET_ELP]

; % -----------------------------now the valid address case------------- %
POP_ASSUM(\thm. ASSUME_TAC ( REWRITE_RULE [] thm ))
THEN POP_ASSUM(\theCase. POP_ASSUM(\thm. MAP_EVERY ASSUME_TAC ( ( [theCase] @ (CONJUNCTS(REWRITE_RULE[PAIR_EQ;theCase] thm)))))
THEN ASSUM_LIST(\asl. REWRITE_TAC[(el 13 asl)])
THEN NORMAL_POP_ASSUM_TAC THEN DELETE_USTEP_TAC 8
THEN NEXT_SYMB_EXEC_TAC 10 THEN DELETE_USTEP_TAC 9
THEN NEXT_SYMB_EXEC_TAC 11 THEN DELETE_USTEP_TAC 10
THEN NEXT_SYMB_EXEC_TAC 12
THEN JMFPOC_POP_ASSUM_TAC THEN DELETE_USTEP_TAC 11

let MSF_TT_FETCH_TAC =
    ASSUM_LIST(\asl. REWRITE_TAC[(el 1 asl);PAIR_EQ]) % 2567.8s %
THEN MSF_CASE_MPC_REWRITE_TAC
THEN NEXT_SYMB_EXEC_TAC 6 THEN DELETE_USTEP_TAC 5
THEN NEXT_SYMB_EXEC_TAC 7 THEN DELETE_USTEP_TAC 6
THEN SYMB_EXEC_ASSUM_TAC 1 T8
% case split based on valid address %
THEN ASSUM_LIST(\asl. ASM_CASES_TAC ( (fst(dest_cond
    (pair_el 9 (snd(dest_eq(snd(dest_thm(el 1 asl)))))))))
THENL
[X--------------------- valid address-------- %
    POP_ASSUM(\theCase. POP_ASSUM(\thm. MAP_EVERY ASSUME_TAC ( ( [theCase] @ (CONJUNCTS
        (REWRITE_RULE [PAIR_EQ; theCase] thm) ))))
THEN DELETE_USTEP_TAC 7
THEN ASSUM_LIST(\asl. ASSUME_TAC % (el 13 asl) is valid_address ... %
    (REWRITE_RULE [PAIR_EQ; update_reg] (el 13 asl)))
THEN ASM_REWRITE_TAC [PAIR_EQ]
% The processor is now stopped due to an addressing exception %
% specialize and rewrite stop_thm show nothing will change %

145
THEN ASSUM_LIST(asl).

let MLC tok = (rator(lhs tok)) = "MacroLevelCycles" ? false in
let endTimeThm = (FIND_ASSUM MLC asl) in
let curTime = (term_to_int (rand(rand(fst(dest_eq(si_(dest_thm(el 2 asl))))))) in
let endTime = (term_to_int (snd(dest_eq(snd(dest_thm(endTimeThm))))) in
ASSUME_TAC(REWRITE_RULE(asl@[sumTHM curTime (endTime-curTime)])
(SPECCL[(int_to_term endTime - curTime); (t_plus_term curTime)])
stop_thm)

THEN POP_ASSUM(thm. REWRITE_TAC[(REWRITE_RULE[PAIR_EQ]thm)])
THEN ASM_REWRITE_TAC[]
THEN REWRITE_TAC[ELP_SET_ELP];

% ------------------------now the valid address case----------------------%
POP_ASSUM(thm. ASSUME_TAC(REWRITE_RULE[]thm))
THEN POP_ASSUM(theCase. POP_ASSUM(thm. MAP_EVERY ASSUME_TAC(  
  (theCase)@ (CONJUNCTS(REWRITE_RULE[PAIR_EQ;theCase]thm)))))
THEN ASSUM_LIST(asl. REWRITE_TAC[(el 13 asl)])
THEN NORMAL_POP_ASSUM_TAC THEN DELETE_USTEP_TAC 7
THEN NEXT_SYMEXEC_TAC 9 THEN DELETE_USTEP_TAC 8
THEN NEXT_SYMEXEC_TAC 10 THEN DELETE_USTEP_TAC 9
THEN NEXT_SYMEXEC_TAC 11 THEN DELETE_USTEP_TAC 10
THEN NEXT_SYMEXEC_TAC 12
THEN JMPPOP_C_Popal_ASSUM_TAC THEN DELETE_USTEP_TAC 11

::
loadf 'digit';
loadf 'decimal';
loadf 'tuple';
loadf 'abstract';

load_parent 'mac_I';

map new_parent ['aux_def';'micro_def';'regs_def';
'aux_thms'; 'time_abs'; 'gen_I'];

let rep_ty = abstract_type 'aux_def' 'opcode';

let ABS_ENV = definition 'macro_def' 'ABS_ENV';
let Opcode = definition 'macro_def' 'Opcode';
let Opc_Val = definition 'macro_def' 'Opc_Val';
let Macro_Int_IMPL_IMP = theorem 'mac_I' 'Macro_Int_IMPL_IMP';

let Micro_state_to_Macro_state = definition 'mac_I'
    'Micro_state_to_Macro_state';

let macro_inst_list = definition 'macro_def' 'macro_inst_list';
let GetMPC = definition 'micro_def' 'GetMPC';
let add_bt? = definition 'micro_def' 'add_bt?';
let Next = definition 'time_abs' 'Next';
let Micro_I = theorem 'micro_aux' 'Micro_I';
let MacroLevelCycles = definition 'mac_I' 'MacroLevelCycles';

let I_rep_ty = abstract_type 'gen_I' 'Impl';

let macro_state = "*:(*wordn#wordn#wordn#wordn#wordn#wordn#memory)";
% a x y p b stop ir ram %

let macro_env = ":(bool)";

let micro_state = "*:(((wordn#wordn#wordn#wordn#memory#bool#bool#bool#
    wordn#bt))";

let micro_env = ":(bool)";

let load_macro_inst = (\x. definition 'macro_def' x);

let macro_defn_list = map load_macro_inst
['NOOP_M'; 'SHR'; 'SHRB'; 'SHL';
 'CMP'; 'WRITEM'; 'WRITEIO'; 'NEG'; 'CALL';
 'READIO'; 'READM'; 'ADDB'; 'ADDS'; 'SUBB';
let load_micro_inst = (\x. theorem 'miczo_def' x);

let Micro_state_to_Macro_state = definition 'mac_I' 'Micro_state_to_Macro_state';

I need some theorems about SUM not provided in the theory

let sum_axiom =
  BETA_RULE (
    REWRITE_RULE [o_DEF] (CONV_RULE (TOP_DEPTH_CONV FUN_EQ_CONV) sum_Axiom));

Some ML function for the inference rules that follow.

let last 1 - (el (length i) i);

let rec term_list_el n l =
  let tm_hd x = rand(fst(dest_comb x)) and
    tm_tl x = snd(dest_comb x) in
    if (n = 0) then tm_hd 1 else
      term_list el (n-l) (tm_tl i) ?
    fail 'term_list_el';;

This is insecure for right now. If anyone is seriously concerned that this isn't right, I'll do it over.

let EL_CONV tm =
  let ((c,n),l) = ((dest_comb#I)o dest_comb) tm in
  let n_int = term_to_int n in
  mk_thm([],"\"tm = \"(term_list_el n_int l)\"\") ?
  failwith 'EL_CONV';;

EL_CONV "EL 3 [0;1;2;3;4;5]";

Some other nice conversions

148
let is_SND_term t =  
    if is_comb t then  
      fst(dest_const(fst(strip_comb t))) = 'SND'  
    else  
      false;;

let SND_CONV t =  
    if is_SND_term t then  
      let op,pr = dest_comb t in  
      let  
        op,[tl;t2] = strip_comb pr in  
      SPECL [tl;t2] (  
        INST_TYPE (((type_of tl),"*:"));  
        ((type_of t2),":**") SND)  
    else  
      failwith 'SND_CDNV';;

let ADD_ASSOC_CONV t =  
    let op1,[t1;t2] = strip_comb t in  
    let op2,[t3;t4] = strip_comb t2 in  
    if op1 = "$+" & op2 = "$+" then  
      SPECL[t1;t3;t4]ADD_ASSOC  
    else fail;;

%  
INV_ADD_ASSOC_CONV "((a+b)+c) --> |- (a+b)+c = a+(b+c)  
%

let INV_ADD_ASSOC = (GEN_ALL o SYM o SPEC_ALL) ADD_ASSOC;;

let INV_ADD_ASSOC_CONV t =  
    let op1,[t1;t2] = strip_comb t in  
    let op2,[t3;t4] = strip_comb t1 in  
    if op1 = "$+" & op2 = "$+" then  
      SPECL[t3;t4;t2] INV_ADD_ASSOC  
    else fail;;

let inv_num_CONV n = (  
    let x,y = dest_comb n in  
    let y_inc = int_to_term ((term_to_int y) + 1) in  
    if not(x = "SUC") then fail else  
    SYM_RULE (num_CONV y_inc))  
? failwith 'inv_num_CONV';;
let instructions = map load_micro_inst

['FETCH_u1'; 'FETCH_u2'; 'FETCH_u3'; 'FETCH_u4';
 'JMP_reqm'; 'JMP_opc'; 'NOOP'; 'SHRS_u1';
 'SHRB_u1'; 'SHLB_u1'; 'AXY_WRITE'; 'SHLS_u1';
 'NO_OVL'; 'NOOP'; 'AXY_WRITE'; 'SHRS_u2';
 'NOOP'; 'AXY_WRITE'; 'SHRB_u2'; 'NOOP';
 'AXY_WRITE'; 'SHLB_u2'; 'NOOP'; 'MF0_u1';
 'MF1_u1';
 'MF2_u1'; 'MF3_u1'; 'MF3_u2'; 'FETCH_u3';
 'MF3_u4'; 'MF3_u5'; 'MF3_u6w1'; 'MF3_u1w4';
 'MF3_u6'; 'MF3_u4'; 'MF3_u5w3'; 'MF3_u6';
 'MF3_u1'; 'MF2_u3'; 'FETCH_u3'; 'MF3_u4';
 'MF3_u5'; 'MF3_u6'; 'COMPARE_u1'; 'WRITEMEM_u1';
 'WRITEIO_u1'; 'NEG_u1'; 'CALL_u1'; 'READIO_u1';
 'READMEM_u1'; 'ADDB_u1'; 'ADDS_u1'; 'SUBB_u1';
 'SUBS_u1'; 'XOR_u1'; 'AND_u1'; 'NOR_u1';
 'ANDMBAR_u1'; 'NOOP'; 'COMPARE_u2'; 'NOOP';
 'WRITEMEM_u2'; 'NOOP'; 'WRITEIO_u2'; 'NOOP';
 'AXY_WRITE'; 'NEGATE_u2'; 'NOOP'; 'CALL_u2';
 'CALL_u3'; 'FETCH_u3'; 'NOOP'; 'READIO_u2';
 'MF3_u5'; 'READIO_u4'; 'NOOP'; 'READIO_u4';
 'CK_VALID_PC'; 'NOOP'; 'ADDB_u2'; 'NOOP';
 'ADDS_u2'; 'CK_VALID_PC'; 'NO_OVL'; 'NOOP';
 'SUBB_u2';
 'NOOP'; 'SUBS_u2'; 'CK_VALID_PC'; 'NO_OVL';
 'NOOP';
 'XOR_u2'; 'NOOP'; 'AND_u2'; 'NOOP';
 'NOR_u2'; 'NOOP'; 'wait_4'; 'wait_3';
 'wait_2'; 'wait_1'; 'MF3_u6'; 'NOOP';
 'NOOP'; 'NOOP'; 'NOOP'; 'NOOP';
 'NOOP'; 'NOOP'; 'NOOP'; 'NOOP';
 'NOOP'; 'NOOP'; 'NOOP'; 'NOOP';
 'NOOP'; 'NOOP'; 'NOOP';
 'NOOP';
];

let micro_inst_list = definition 'micro_def' 'micro_inst_list';

%--------------------------------------------------------------------------
% Beginning of MPC
%--------------------------------------------------------------------------%
let FETCH_ADDR = "(F,F,F,F,F,F)";%

%--------------------------------------------------------------------------
% Offset into microrom lookup table may need to change
%--------------------------------------------------------------------------%

150
let OFFSET = "4";;

Using MK_Micro_Int_Inst_LEMMA, we can prove a lemma of the form

\[
! \text{Micro_Int}
\]

for every microinstruction, by simply giving its position in the list. Mapping the inference rule onto a list of integers from 0 to 127 yields a list of lemmas for each micro instruction. The entire process (exclusive of autoloading time) takes < 700 sec.

let Micro_Int_SPEC =
  PURE_ONCE_REWRITE_RULE [micro_inst_list;GetMPC] (  
  BETA_RULE (  
    SPECL ["rep:"rep_ty";  
      "\t. (reg t,m t,ins t,din t,dout t,ram t,b t,stop t,ovl t,m t,res t,mpc t)):time->'micro_state";  
      "\t. (reset_e t)):time->'micro_env"") Micro_I));

let MK_Micro_Int_Inst_LEMMA inst =
  let tp = mk_n_tuple_from_int 7 inst in
  let mpc_term = "mpc t = \t. (reg t,m t,ins t,din t,dout t,ram t,b t,stop t,ovl t,m t,res t,mpc t):time->'micro_state";
  "\t. (reset_e t)):time->'micro_env"") Micro_I));

let MK_Micro_Int_Inst_LEMMA inst =
  let tp = mk_n_tuple_from_int 7 inst in
  let mpc_term = "mpc t = \t. (reg t,m t,ins t,din t,dout t,ram t,b t,stop t,ovl t,m t,res t,mpc t):time->'micro_state";
  "\t. (reset_e t)):time->'micro_env"") Micro_I));

let MK_Micro_Int_Inst_LEMMA inst =
  let tp = mk_n_tuple_from_int 7 inst in
  let mpc_term = "mpc t = \t. (reg t,m t,ins t,din t,dout t,ram t,b t,stop t,ovl t,m t,res t,mpc t):time->'micro_state";
  "\t. (reset_e t)):time->'micro_env"") Micro_I));

let MK_Micro_Int_Inst_LEMMA inst =
  let tp = mk_n_tuple_from_int 7 inst in
  let mpc_term = "mpc t = \t. (reg t,m t,ins t,din t,dout t,ram t,b t,stop t,ovl t,m t,res t,mpc t):time->'micro_state";
  "\t. (reset_e t)):time->'micro_env"") Micro_I));
"stop t:bool";
"ovl t:bool";
"mar t:*address";
"res t:*wordn";

let mk_num_list n
  = letrec mk_num_list_aux n m =
      if n = m then [m] else
        (n, (mk_num_list_aux (n+1) m)) in
      mk_num_list_aux 0 n;;

let Micro_Int_Inst_list = map MK_Micro_Int_Inst_LEMMA (mk_num_list 32);;
% let Micro_Int_Inst_list = map MK_Micro_Int_Inst_LEMMA (mk_num_list 127);;
%
% correct up to here
%

% Normalize top assumption (get rid of add_bt7)
 ...
 let NORMAL_POP_ASSUM_TAC =
   POP_ASSUM (\thm. ASSUME_TAC (  
       CONV_RULE (ONCE_DEPTH_CONV bt7_iival_CONV) (  
         CONV_RULE DEC_ADD_CONV (  
           % DEC_ADD_CONV broken for "0 + 1" %  
           PURE_ONCE_REWRITE_RULE [ADD_CLAUSES] (  
             CONV_RULE (ONCE_DEPTH_CONV bt7_val_CONV) (  
               REWRITE_RULE [add bt7 thm]))))));;
let RANGE_LEMMA = TAC_PROOF
(([],"!t1 t2 (mpc:\text{time}\to\text{bt7}) x .
  (!'t', t1 < t' \land \ t' < t2 \implies \ ~(mpc t' = x)) \land
  (mpc t2 = x) \implies
  (!'t', t1 < t' \land \ t' < (t2 + 1) \implies \ ~(mpc t' = x))")
REPEAT STRIP_TAC
THEN ASSUM_LIST (asl. ASSUME_TAC (SPEC "t':\text{time}" (el 5 asl)))
THEN ASSUM_LIST (asl. STRIP_ASSUME_TAC (REWRITE_RULE [SYM_RULE ADDI;LESS_THM] (el 3 asl)))
THENL [ASSUM_LIST (asl. ASSUME_TAC (REWRITE_RULE [el 1 asl] (el 3 asl)))
  ALL_TAC]
THEN RES_TAC
);

let LESS_SQUEEZE_LEMMA =
let LESS_EQ_SUC =
  let PURE_ONCE_REWRITE_RULE [DISJ_SYM] LESS_THM in
  PURE_ONCE_REWRITE_RULE [ADDI] (PURE_ONCE_REWRITE_RULE [LESS_EQ_SUC] (PURE_DNCE_REWRITE_RULE [LESS_OR_EQ] LESS_EQ_ANTISYM));;

let Macro_Int_IMPL_IMP_LEMMA =
  let (INST_LOOP_TAC tm_init):tactic =
    let is_begin than =
      snd(dest_eq thm) = FETCH_ADDR in
    let tuple_val than =
      term_to_int(bt_val_func(snd(dest_eq (hm))) _n
    letrec INST_LOOP_TAC_AUX tm ((asl,w):goal) =
      let INST_TAC n =
        IMP_RES_TAC (el n Micro_Int_Inst_list) THEN

153
ASSUM_LIST
C0NJUNCTS

(\x.

REWRITE_RULE
=

let

n

let

gl,p

let

(asl',_')

[PAIR_EQ]

(tuple_val
=

let

gll,pl

if

(is_begin

(el

INST_TAC
=

=

(hd

split

1 x))))

(asl,w)

+

in

i in

in

in

(

(el

I asl'))

then

tm)

else

(EXISTS_TAC

(INST_LOOP_TAC_AUX
gll,(p

(el

gl)

map

gl

"('tm)÷1")

o mapshape(map

INST_LOOP_TAC_AUX

let

ASSUME_TAC

I asl))

n

map

(flat

MAP_EVERY

(

gl)

length

"('tm_init

DECODE_M_CORRECTLY_IMP

in

gll)pl))
+

zn

1)";;

= new_definition

('DECODE_M_CORRECTLY_IMP',
"DEOODE_M_CORRECTLY_IMP
! (ins:.wordn)
let

ins_dec

let

opc=

let

mem_req

(b:bool)
=

(decode

(FST

(SND

=

let
(((opc=

(rep:'rep_ty)

dec

.
rep

(SND

stop

=

(opc=

(F,F,F,F,T))

\/

(opc=

(F,F,F,T,F))

\/

(F,F,F,T,T))

\/

(opc=
((opt

(F,F,T,F,F)))
=

((mem_req

let

=>

(T,T,T,T,T))
=

T)

/\

=>

let

inst

=

ins_dec)

in

in

((mem_req

(dec_stop

=

n

b))

in

((mem_req

MX_INST_CORRECT_GOAL

ins,

in

(FST
\/

=

rep

ins_dec))

(F,F,F,F,F))

(opc

(opcode

ins_dec))

(SND

=

= F)
=

F)

/\

/\

(dec_stop

(dec_stop

=

F))

= T))

[

F))))");;

=

term_list_el

n

(snd(dest_eq(
snd(dest_forall(con¢l
"!(rep:'rep__y)
(b stop

macro_inst_list)))))

(regs:time->(*mordn)list)

ovl:time->bool)

(m

ins

(mar:time->*address)

din

zn
dout:time->*wordn)

(res:time->*wordn)

(ram:time->.memory)

(mpc:time->btT)

(reset_e:time->bool).
DEC0DE_M_CORRECTLY_IMP

rep

(Macro_Int_IMPL_IMP
(\t.
(\t.

let
"'

((Micro_I
(\t.

154

reg

rep

t,m

reset_e

t,ins
t)

= prove_thm

stop_thm
(n:num)

(reg

==>

t,din

t,dout

t,

ram

t,b

t,stop

t,ovl

"inst)";;

('stop_thm',

(t:num).
(rep:_rep_ty)
t,m

t,

ins

t,

din

t,

dout

t,

ram

t,

b

t,

stop

t,

ovl

t,

t,

mar

t,

res

t,

mpc

t))


map (delete_cache o fst)(cached_theories());

let T_PLUS_Z_LEMM_ = TAC_PROOF(([], "! t.t+Z=((((((t+l)+l)+1)+1)+1)+1)+1+1"), GEN_TAC
THEN REPEAT (PURE_ONCE_REWRITE_TAC [SYM_RULE ADD_ASSOC]
THEN DEC_ADD_TAC));
Appendix E: MICRO LEVEL SPECIFICATION

File: ucode_aux.ml

Description: Defines the ML functions and constants necessary to describe the microinstructions. This file is loaded by several files that draft theories.

Modified by ETS:

Includes new wait microinstruction labels
Removed seq control case stop_ovl_ill_pdest
This case can be simulated by using
stop_ovl and stop_ill_pdest.
Added stop_pcwrite.

The functional representation of a microinstruction:
(address, seq_alu_ctl(seq, alu), dec_ctl(sig), mem(op),
srcdst(rfc, dfc, rfsel, dfsel), enable(copy), select(addout, datain, mout))

The possible values of various arguments is as follows: (X = don't care)
address - symbol / X
seq  - idle / jmp / opcjmp / jmp / stop_ovl / stop_ill_addr /
     stop_ill_pdest / stop_pcwrite
alu  - mthrd (or idle) / rthrd / compare / negate / add_bcarry /
      add / sub_bcarry / sub / xor / and / nor / and_not /
      shr_s / shr_b / shl_s / shl_b
sig   - inhibit / allow
op    - idle / rio / rmem / wio / wmen
rfc   - inst_rf (or X) / m_rf
dfc   - inst_df (or X) / m_df
rfsel - regA (or X) / regX / regY / regP
dfsel - regA (or X) / regX / regY / regP / regM / regADDR
copy  - none / data / res / both
addrout = p (or X) / addr

datain = m (or x) / ins

mout = m (or X) / one / addr

%-----------------------------------------------------------------------

%-----------------------------------------------------------------------

Definition of labels in microcode

-----------------------------------------------------------------------%

let X7 = "(F,F,F,F,F,F,F)" ;


let noop = "(F,F,F,F,T,T,F)" ;

let shrsl = "(F,F,F,T,T,T,F)" ;

let shrbl = "(F,F,T,F,F,F,T)" ;

let shlbl = "(F,F,T,F,T,F,F)" ;

let mfO = "(F,F,T,F,T,T,T)" ;

let mr01 = "(F,T,F,F,F,F,F)" ;

let mfll = "(F,T,F,F,F,T,F)" ;

let mr21 = "(F,T,F,F,T,F,T)" ;

let le_, base = "(F,T,F,F,T,T,F)" ;

let comparel = "(F,T,T,T,T,F,F)" ;

let writememl = "(F,T,T,T,T,F,T)" ;

let writeiol = "(F,T,T,T,T,T,T)" ;

let negl = "(T,F,F,F,F,F,T)" ,

let calll = "(T,F,F,F,T,F,F)" ;

let readiol = "(T,F,F,T,F,F,F)" ;

let readmeml = "(T,F,F,T,T,F,F)" ;

let addbl = "(T,F,F,T,T,T,T)" ;

let addsl = "(T,F,T,F,F,F,T)" ;

let subbl = "(T,F,T,F,T,F,T)" ;

let subs1 = "(T,F,T,F,T,T,T)" ;

let subs1 = "(T,F,T,F,T,T,T)" ;
let xor1 = "(T,F,T,F,T,T)";;
let and1 = "(T,F,T,T,F,T)";;
let nor1 = "(T,F,T,T,T,F,T)";;
let wait_1 = "(T,F,F,F,T,F,F)";;
let wait_2 = "(T,T,F,F,T,T,T)";;
let wait_3 = "(T,T,F,F,F,T,F)";;
let wait_4 = "(T,T,F,F,F,F,T)";;
let X = 0;; %---- dont care ------%  
let idle = 0;; %------ idle -------%

Definition of control signals for microsequencing logic
................................................................
idle = 0

let mjmp = 1;;
let opcjmp = 2;;
let jmp = 3;;
let stop_ovl = 4;;
let stop_ill_addr = 5;;
let stop_ill_pdest = 6;;
let stop_pcwrite = 7;;

let seq_ctl x =
  (x = idle) => "(F, F, F)" |
  (x = mjmp) => "(F, F, T)" |
  (x = opcjmp) => "(F, T, F)" |
  (x = jmp) => "(F, T, T)" |
  (x = stop_ovl) => "(T, F, F)" |
  (x = stop_ill_addr) => "(T, F, T)" |
  (x = stop_ill_pdest) => "(T, T, F)" |
  "(T, T, T)";;

Definition of control signals for alu
................................................................

%---------------------------------------------------------------------
% idle = 0 %

let mjmp = 1;;
let opcjmp = 2;;
let jmp = 3;;
let stop_ovl = 4;;
let stop_ill_addr = 5;;
let stop_ill_pdest = 6;;
let stop_pcwrite = 7;;

let seq_ctl x =
  (x = idle) => "(F, F, F)" |
  (x = mjmp) => "(F, F, T)" |
  (x = opcjmp) => "(F, T, F)" |
  (x = jmp) => "(F, T, T)" |
  (x = stop_ovl) => "(T, F, F)" |
  (x = stop_ill_addr) => "(T, F, T)" |
  (x = stop_ill_pdest) => "(T, T, F)" |
  "(T, T, T)";;

%---------------------------------------------------------------------

% Definition of control signals for alu
................................................................

%---------------------------------------------------------------------

% idle = 0 %
% idle = 0 %

let mthro = 0;;
let rthro = 1;;
let compare = 2;;
let negate = 3;;
let add_bcarry = 4;;
let add = 5;;
let sub_bcarry = 6;;
let sub = 7;;
let xor = 8;;
let and = 9;;
let nor = 10;;
let and_not = 11;;
let shr_s = 12;;
let shr_b = 13;;
let shl_s = 14;;
let shl_b = 15;;

let alu_ctl x =
    (x = idle) => "(F, F, F, F)" |
    (x = mthro) => "(F, F, F, F)" |
    (x = rthro) => "(F, F, F, T)" |
    (x = compare) => "(F, F, T, F)" |
    (x = negate) => "(F, F, T, T)" |
    (x = add_bcarry) => "(F, T, F, F)" |
    (x = add) => "(F, T, F, T)" |
    (x = sub_bcarry) => "(F, T, T, F)" |
    (x = sub) => "(F, T, T, T)" |
    (x = xor) => "(T, F, F, F)" |
    (x = and) => "(T, F, F, T)" |
    (x = nor) => "(T, F, T, F)" |
    (x = and_not) => "(T, F, T, T)" |
    (x = shr_s) => "(T, F, F, F)" |
    (x = shr_b) => "(T, T, F, F)" |
(x = shl_s) => "(T, T, F)" | "(T, T, T)" ;;

let seq_alu_ctl (seq, alu) = "((seq_ctl seq), (alu_ctl alu))";;

%---------------------------------------------------------------------------
Definition of decoder control signal
---------------------------------------------------------------------------% 

let inhibit = "F";;
let allow = "T";;
let dec_ctl (sig) = "sig";;

%---------------------------------------------------------------------------
Definition of memory control signals
---------------------------------------------------------------------------% 

let rio = 1;;
let rmem = 2;;
let wio = 3;;
let wmem = 4;;

let mem (op) =
  (op = idle) => "(F,F,F)" | 
  (op = rio) => "(T,F,T)" | 
  (op = rmem) => "(T,F,F)" | 
  (op = wio) => "(F,T,T)" | 
  "(F,T,F)";;

%---------------------------------------------------------------------------
Definition of source & destination reg select lines
---------------------------------------------------------------------------% 

let regA = 0;;
let regX = 1;;
let regY = 2;;
let regP = 3;;
let regM = 6;;
let regADDR = 7;;
let inst_rf = 0;;

let m_rf = 1;;

let inst_df = 0;;

let m_df = 1;;

let rf x =
  ((x = regA) or (x = X)) => "(F, F)" |
  (x = regX) => "(F, T)" |
  (x = regY) => "(T, F)" |
  "(T, T)";;

let df x =
  ((x = regA) or (x = X)) => "(F, F, F)" |
  (x = regX) => "(F, F, T)" |
  (x = regY) => "(F, T, F)" |
  (x = regP) => "(F, T, T)" |
  (x = regM) => "(T, T, F)" |
  (x = regADDR) => "(T, T, T)" |
  "(F, T, T)";;  % P register %

let srcdat (rfc, dfc, rfssel, dfssel) =
  "((rf rfssel), (df dfssel),
   "((rfc = m_rf) => "T" | "F"),
   "((dfc = m_df) => "T" | "F"))";;

let none = 0;;

let data = 1;;

let res = 2;;

let both = 3;;

let enable (x) =
  (x = none) => "(F, F)" |
  (x = data) => "(T, F)" |
  (x = res) => "(F, T)" |
  "(T, T)";;

let p = 0;;

let m = 0;;

let ins = 1;;
let one = 1;;

let addr = 2;;

let whichm x =
  (x = m) => "(F, F)" |
  (x = one) => "(F, T)" |
  (x = addr) => "(T, F)" |
  "(T, T)";;

let select (addrout, datain, mout) =
  "((addrout = addr) => "T" | "F"),
  "((datain = ins) => "T" | "F"),
  "(whichm mout))";;
%--- % %
%
File: def_uinst.ml

Description: Defines the microinstructions and microrom for the
micro-level.

Modifications by ETS

Include new wait, NO_PC_WRITE and CK_VALID_PC, NO_OVL microinstructions.
Replaced SHLS_u3_mc with NO_OVL fby CK_VALID_PC
Logical operations' semantics now stop on write to pc
Reorganized microcode slightly

NO_PC_WRITE changed to AXW_WRITE (i/o space and memory also invalid)

set_search_path (search_path() @ lib_dir_list);;

system '/bin/rm uinst.th';
loadf 'ucode_aux';;
new_theory 'uinst';;

%----------------------------------

If you change these addresses, change the list in ucode_aux.ml as well.

let wait_0 = "(T,T,F,F,T,F,T)"; is not in ucode_aux.ml

%----------------------------------

Definition of labels in microcode

%----------------------------------

let X7 = "(F,F,F,F,F,F,F)";;
let fetch = "(F,F,F,F,F,F,F)";;
let noop = "(F,F,F,F,T,T,F)";;
let shrsl = "(F,F,F,T,T,T,F)";;
let shrbl = "(F,F,T,F,F,F,T)";;
let shrbl = "(F,F,T,F,T,F,F)";;
let mf0 = "(F,F,T,F,T,T,T)";;
let mf01 = "(F,T,F,F,F,F,F)";;
let mf21 = "(F,T,F,F,T,T)";;
let base = "(F,T,F,F,T,T)";;
let compare1 = "(F,T,F,F,F,T)";;
let writemem1 = "(F,T,T,T,F,T)";;
let writeio1 = "(F,T,T,T,T,T)";;
let neg1 = "(T,F,F,F,F,T)";;
let call1 = "(T,F,F,F,F,F)";;
let readio1 = "(T,F,F,F,F,F)";;
let readmem1 = "(T,F,F,T,T,F)";;
let addb1 = "(T,F,F,T,T,T)";;
let addsl = "(T,F,T,F,F,F)";;
let subb1 = "(T,F,T,F,F,F)";;
let subsl = "(T,F,T,F,T,F)";;
let xor1 = "(T,F,T,F,F,F)";;
let and1 = "(T,F,F,F,F,F)";;
let nor1 = "(T,F,F,F,F,F)";;
let wait_0 = "(T,T,F,F,T,F)";;
let wait_1 = "(T,T,F,F,F,F)";;
let wait_2 = "(T,T,F,F,F,F)";;
let wait_3 = "(T,T,F,F,F,F)";;
let wait_4 = "(T,T,F,F,F,F)";;

%---- added by ETS -----%
let AX Y_WRITE_mc = new_definition
  ('AX Y_WRITE_mc c ,
   "AX Y_WRITE_mc =
    ('X7,  '(seq_alu_ctl(stop_pcwrite, idle)),  '(dec_ctl(inhibit)),
     ́(mem(idle)),  ́(srcdst(X,X,X,X)),  ́(enable(none)),
     ́(select(X, X, X)))"
  );;
let CK_VALID_PC_mc = new_definition
('CK_VALID_PC_mc',
"CK_VALID_PC_mc =
('X7, "(seq_alu_ctl(stop_ill_pdest, idle)), "(dec_ctl(inhibit)),
"(mem(idle)), "(srcdst(X,X,X,X)), "(enable(none)),
"(select(X, X, X)))"
);

let NO_OVL_mc = new_definition
('NO_OVL_mc',
"NO_OVL_mc =
('X7, "(seq_alu_ctl(stop_ovl, idle)), "(dec_ctl(inhibit)),
"(mem(idle)), "(srcdst(X,X,X,X)), "(enable(none)),
"(select(X, X, X)))"
);

%%%%%-----old stuff -------%%%%% 

let FETCH_u1_mc = new_definition
('FETCH_u1_mc',
"FETCH_u1_mc =
('X7, "(seq_alu_ctl(idle, idle)), "(dec_ctl(inhibit)),
"(mem(rmem)), "(srcdst(X,X,X,X)), "(enable(none)), "(select(p,X,X)))"
);

let FETCH_u2_mc = new_definition
('FETCH_u2_mc',
"FETCH_u2_mc =
('X7, "(seq_alu_ctl(idle, add)), "(dec_ctl(inhibit)), "(mem(idle)),
"(srcdst(m_rf,m_df,regP,regP)), "(enable(res)), "(select(X, X, one)))"
);

let FETCH_u3_mc = new_definition
('FETCH_u3_mc',
"FETCH_u3_mc =
('X7, "(seq_alu_ctl(stop_ill_addr, idle)), "(dec_ctl(inhibit)),
"(mem(idle)), "(srcdst(X,X,X,X)), "(enable(none)),
"(select(X, X, X)))"
);

let FETCH_u4_mc = new_definition
('FETCH_u4_mc',
"FETCH_u4_mc =
('X7, "(seq_alu_ctl(idle, idle)), "(dec_ctl(inhibit)), "(mem(idle)),
"(srcdst(X,X,X,X)), "(enable(data)), "(select(X, ins, X)))"
)
let JMP_reqm_mc = new_definition
    ('JMP_reqm_mc',
    "JMP_reqm_mc =
    ("mfO, "(seq_alu_ctl(mjmp, idle)), "(dec_ctl(allow)),
    "(mem(idle)), "(srcdst(X,X,X,X)), "(enable(none)), "(select(X, X, X))")"
    );;

let JMP_opc_mc = new_definition
    ('JMP_opc_mc',
    "JMP_opc_mc =
    ("noop, "(seq_alu_ctl(opcjmp, idle)), "(dec_ctl(inhibit)),
    "(mem(idle)), "(srcdst(X,X,X,X)), "(enable(none)), "(select(X, X, X))")"
    );;

let NOOP_mc = new_definition
    ('NOOP_mc',
    "NOOP_mc =
    ("fetch, "(seq_alu_ctl(jmp, idle)), "(dec_ctl(inhibit)),
    "(mem(idle)), "(srcdst(X,X,X,X)), "(enable(none)), "(select(X, X, X))")"
    );;

let SHRS_u1_mc = new_definition
    ('SHRS_u1_mc',
    "SHRS_u1_mc =
    ("shrsl, "(seq_alu_ctl(jmp, idle)), "(dec_ctl(inhibit)),
    "(mem(idle)), "(srcdst(X,X,X,X)), "(enable(none)), "(select(X, X, X))")"
    );;

let SHRB_u1_mc =
    'SHRB_u1_mc',
    "SHRB_u1_mc =
    ("shrbl, "(seq_alu_ctl(jmp, idle)), "(dec_ctl(inhibit)),
    "(mem(idle)), "(srcdst(X,X,X,X)), "(enable(none)), "(select(X, X, X))")"
    );;

let SHLS_u1_mc = new_definition
    ('SHLS_u1_mc',
    "SHLS_u1_mc =
    ("X7, "(seq_alu_ctl(idle, shl_s)), "(dec_ctl(inhibit)), "(mem(idle)),
    "(srcdst(inst_rf,inst_df,X,X)), "(enable(res)), "(select(X, X, X))")"
    );;

let SHLB_u1_mc = new_definition
    ('SHLB_u1_mc',
    "SHLB_u1_mc =
    "(seq_alu_ctl(idle, shl_s)), "(dec_ctl(inhibit)), "(mem(idle)),
    "(srcdst(inst_rf,inst_df,X,X)), "(enable(res)), "(select(X, X, X))")"
    );;
(`shlb1, `seq_alu_ctl(jmp, idle)), `dec_ctl(inhibit)),
  `mem(idle)), `srcdst(X,X,X,X)), `enable(none)), `select(X, X, X))"
);

let SHLB_u2_mc = new_definition
  (`SHLB_u2_mc',
   "SHLB_u2_mc =
   (`X7, `seq_alu_ctl(idle, shl_b)), `dec_ctl(inhibit)),
   `mem(idle)),
   `srcdst(inst_rf,inst_df,X,X)), `enable(res)), `select(X, X, X))"
);

let SHRS_u2_mc = new_definition
  (`SHRS_u2_mc',
   "SHRS_u2_mc =
   (`X7, `seq_alu_ctl(idle, shr_s)), `dec_ctl(inhibit)), `mem(idle)),
   `srcdst(inst_rf,inst_df,X,X)), `enable(res)), `select(X, X, X))"
);

let SHRB_u2_mc = new_definition
  (`SHRB_u2_mc',
   "SHRB_u2_mc =
   (`X7, `seq_alu_ctl(idle, shr_b)), `dec_ctl(inhibit)), `mem(idle)),
   `srcdst(inst_rf,inst_df,X,X)), `enable(res)), `select(X, X, X))"
);

let MFO_u1_mc = new_definition
  (`MFO_u1_mc',
   "MFO_u1_mc =
   (`mf01, `seq_alu_ctl(jmp, idle)), `dec_ctl(inhibit)),
   `mem(idle)), `srcdst(X,X,X,X)), `enable(none)), `select(X, X, X))"
);

let MF1_u1_mc = new_definition
  (`MF1_u1_mc',
   "MF1_u1_mc =
   (`mf11, `seq_alu_ctl(jmp, idle)), `dec_ctl(inhibit)),
   `mem(idle)), `srcdst(X,X,X,X)), `enable(none)), `select(X, X, X))"
);

let MF2_u1_mc = new_definition
  (`MF2_u1_mc',
   "MF2_u1_mc =
   (`mf21, `seq_alu_ctl(jmp, idle)), `dec_ctl(inhibit)),
   `mem(idle)), `srcdst(X,X,X,X)), `enable(none)), `select(X, X, X))"
);
let MF3_u1_mc = new_definition
  ('MF3_u1_mc',
   "MF3_u1_mc =
   ("X7", "(seq_alu_ctl(idle, mtho)), "(dec_ctl(inhibit)), "(mem(idle)),
     "(srcdst(X,m_df,X,regM)), "(enable(res)), "(select(X, X, addr)))"
   );

let MF3_u2_mc = new_definition
  ('MF3_u2_mc',
   "MF3_u2_mc =
   ("X7", "(seq_alu_ctl(idle, add)), "(dec_ctl(inhibit)), "(mem(idle)),
     "(srcdst(m_rf,m_df,regY,regADDK)), "(enable(res)), "(select(X, m))"
   );

let MF3_u4_mc = new_definition
  ('MF3_u4_mc',
   "MF3_u4_mc =
   ("X7", "(seq_alu_ctl(idle, idle)), "(dec_ctl(inhibit)), "(mem(rmem)),
     "(srcdst(X,X,X,X)), "(enable(data)), "(select(X, m, X))"
   );

let MF3_u5_mc = new_definition
  ('MF3_u5_mc',
   "MF3_u5_mc =
   ("X7", "(seq_alu_ctl(idle, idle)), "(dec_ctl(inhibit)),
     "(mem(idle)), "(srcdst(X,X,X,X)), "(enable(none)), "(select(X, m, X))"
   );

let MF3_u6_mc = new_definition
  ('MF3_u6_mc',
   "MF3_u6_mc =
   ("base,"(seq_alu_ctl(opcjmp,idle)), "(dec_ctl(inhibit)),
     "(mem(idle)), "(srcdst(X,X,X,X)), "(enable(none)), "(select(X, X, X))"
   );

let MF2_u3_mc = new_definition
  ('MF2_u3_mc',
   "MF2_u3_mc =
   ("X7", "(seq_alu_ctl(idle, add)), "(dec_ctl(inhibit)), "(mem(idle)),
     "(srcdst(m_rf,m_df,regX,regADDR)), "(enable(res)), "(select(X, X, m))"
   );

let COMPARE_u1_mc = new_definition
  ('COMPARE_u1_mc',
   "COMPARE_u1_mc =
   ("compare1,"(seq_alu_ctl(jmp, idle)), "(dec_ctl(inhibit)),
   "(mem(idle)), "(srcdst(X,X,X,X)), "(enable(data)), "(select(X, m, X))"
   );

169
let WRITEMEM_uI_mc = new_definition
('WRITEMEM_uI_mc',
"WRITEMEM_uI_mc =
('writemem1,'(seq_alu_ctl(jmp, idle)),'(dec_ctl(inhibit)),
"(mem(idle)), "(srcdst(X,X,X,X)), "(enable(none)), "(select(X, X, X)))"
);

let WRITEIO_uI_mc = new_definition
('WRITEIO_uI_mc',
"WRITEIO_uI_mc =
('writeiol,'(seq_alu_ctl(jmp, idle)),'(dec_ctl(inhibit)),
"(mem(idle)), "(srcdst(X,X,X,X)), "(enable(none)), "(select(X, X, X)))"
);

let NEG_uI_mc = new_definition
('NEG_uI_mc',
"NEG_uI_mc =
('negl, "(seq_alu_ctl(jmp, idle)), "(dec_ctl(inhibit)),
"(mem(idle)), "(srcdst(X,X,X,X)), "(enable(none)), "(select(X, X, X)))"
);

let CALL_uI_mc = new_definition
('CALL_uI_mc',
"CALL_uI_mc =
('calll, "(seq_alu_ctl(jmp, idle)), "(dec_ctl(inhibit)),
"(mem(idle)), "(srcdst(X,X,X,X)), "(enable(none)), "(select(X, X, X)))"
);

let READIO_uI_mc = new_definition
('READIO_uI_mc',
"READIO_uI_mc =
('readiol,'(seq_alu_ctl(jmp, idle)), "(dec_ctl(inhibit)),
"(mem(idle)), "(srcdst(X,X,X,X)), "(enable(none)), "(select(X, X, X)))"
);

let READMEM_uI_mc = new_definition
('READMEM_uI_mc',
"READMEM_uI_mc =
('readmeml,'(seqalu_ctl(jmp, idle)),'(dec_ctl(inhibit)),
"(mem(idle)), "(srcdst(X,X,X,X)), "(enable(none)), "(select(X, X, X)))"
);

let ADDB_uI_mc = new_definition
('ADDB_uI_mc',
"ADDB_uI_mc =
"(mem(idle)), "(srcdst(X,X,X,X)), "(enable(none)), "(select(X, X, X))""
);
let ADDS_u1_mc = new_definition
('ADDS_u1_mc',
"ADDS_u1_mc =
("addb1, "(seq_alu_ctl(jmp, idle)), "(dec_ctl(inhibit)),
  "(mem(idle)), "(srcdst(X,X,X,X)), "(enable(none)), "(select(X, X, X))")"
);

let SUBB_u1_mc = new_definition
('SUBB_u1_mc',
"SUBB_u1_mc =
("subb1, "(seq_alu_ctl(jmp, idle)), "(dec_ctl(inhibit)),
  "(mem(idle)), "(srcdst(X,X,X,X)), "(enable(none)), "(select(X, X, X))")"
);

let SUBS_u1_mc = new_definition
('SUBS_u1_mc',
"SUBS_u1_mc =
("subsl, "(seq_alu_ctl(jmp, idle)), "(dec_ctl(inhibit)),
  "(mem(idle)), "(srcdst(X,X,X,X)), "(enable(none)), "(select(X, X, X))")"
);

let XOR_u1_mc = new_definition
('XOR_u1_mc',
"XOR_u1_mc =
("xorl, "(seq_alu_ctl(jmp, idle)), "(dec_ctl(inhibit)),
  "(mem(idle)), "(srcdst(X,X,X,X)), "(enable(none)), "(select(X, X, X))")"
);

let AND_u1_mc = new_definition
('AND_u1_mc',
"AND_u1_mc =
("andl, "(seq_alu_ctl(jmp, idle)), "(dec_ctl(inhibit)),
  "(mem(idle)), "(srcdst(X,X,X,X)), "(enable(none)), "(select(X, X, X))")"
);

let NOR_u1_mc = new_definition
('NOR_u1_mc',
"NOR_u1_mc =
("norl, "(seq_alu_ctl(jmp, idle)), "(dec_ctl(inhibit)),
  "(mem(idle)), "(srcdst(X,X,X,X)), "(enable(none)), "(select(X, X, X))")"
);

let ANDMBAR_u1_mc = new_definition
('ANDMBAR_u1_mc',
"ANDMBAR_u1_mc =
("addl, "(seq_alu_ctl(jmp, idle)), "(dec_ctl(inhibit)),
  "(mem(idle)), "(srcdst(X,X,X,X)), "(enable(none)), "(select(X, X, X))")"
);
"ANDMEBAR_u1_mc =
("X7", "seq_alu_ctl(stop_pmwite, and_not)", "dec_ctl(inhibit)", "mem(idle)",
"(srcdst(inst_rf, inst_df, X, X))", "enable(res)", "select(X, X, m))")
);

let COMARE_u2_mc = new_definition
("COMARE_u2_mc",
"COMARE_u2_mc =
("X7", "seq_alu_ctl(idle, compare)", "dec_ctl(inhibit)", "mem(wmem)",
"(srcdst(inst_rf, X, X)), "enable(none)", "select(X, X, m))")
);

let WRITEMEM_u2_mc = new_definition
("WRITEMEM_u2_mc",
"WRITEMEM_u2_mc =
("X7", "seq_alu_ctl(idle, idle)", "dec_ctl(inhibit)", "mem(wmem)",
"(srcdst(inst_rf, X, X)), "enable(none)", "select(X, X, m))")
);

let WRITEIO_u2_mc = new_definition
("WRITEIO_u2_mc",
"WRITEIO_u2_mc =
("X7", "seq_alu_ctl(idle, idle)", "dec_ctl(inhibit)", "mem(wio)",
"(srcdst(inst_rf, X, X, X)), "enable(none)", "select(addr, X, X))")
);

let NEGATE_u2_mc = new_definition
("NEGATE_u2_mc",
"NEGATE_u2_mc =
("X7", "seq_alu_ctl(idle, negate)", "dec_ctl(inhibit)", "mem(idle)",
"(srcdst(X, inst_df, X, X)), "enable(res)", "select(X, X, m))")
);

let CALL_u2_mc = new_definition
("CALL_u2_mc",
"CALL_u2_mc =
("X7", "seq_alu_ctl(idle, rthro)", "dec_ctl(inhibit)", "mem(idle)",
"(srcdst(m_rf, m_df, regP, regY)), "enable(res)", "select(X, X, X))")
);

let CALL_u3_mc = new_definition
("CALL_u3_mc",
"CALL_u3_mc =
("X7", "seq_alu_ctl(idle, mthro)", "dec_ctl(inhibit)", "mem(idle)",
"(srcdst(X, m_df, X, regP)), "enable(res)", "select(X, X, m))")
);

let READIO_u2_mc = new_definition
172
('READIO_u2_mc',
  "READIO_u2_mc =
  ('X7, "(seq_alu_ctl(idle, idle)), "(dec_ctl(inhibit)),
   "(mem(rio)), "(srcdst(X,X,X,X)), "(enable(none)), "(select(addr, X, X)))"
 );

let READIO_u4_mc = new_definition
('READIO_u4_mc',
  "READIO_u4_mc =
  ('X7, "(seq_alu_ctl(stop_pcwrite, mthro)),
   "(dec_ctl(inhibit)), "(mem(idle)),
   "(srcdst(X,inst_df,X,X)), "(enable(res)), "(select(X, X, m)))"
 );

let READMEM_u2_mc = new_definition
('READMEM_u2_mc',
  "READMEM_u2_mc =
  ('X7, "(seq_alu_ctl(idle, mthro)), "(dec_ctl(inhibit)), "(mem(idle)),
   "(srcdst(X,inst_df,X,X)), "(enable(res)), "(select(X, X, m)))"
 );

let ADDB_u2_mc = new_definition
('ADDB_u2_mc',
  "ADDB_u2_mc =
  ('X7, "(seq_alu_ctl(stop_pcwrite, add_bcarry)),
   "(dec_ctl(inhibit)), "(mem(idle)),
   "(srcdst(inst_rf,inst_df,X,X)), "(enable(res)), "(select(X, X, m)))"
 );

let ADDS_u2_mc = new_definition
('ADDS_u2_mc',
  "ADDS_u2_mc =
  ('X7, "(seq_alu_ctl(idle, add)), "(dec_ctl(inhibit)), "(mem(idle)),
   "(srcdst(inst_rf,inst_df,X,X)), "(enable(res)), "(select(X, X, m)))"
 );

let SUBB_u2_mc = new_definition
('SUBB_u2_mc',
  "SUBB_u2_mc =
  ('X7, "(seq_alu_ctl(stop_pcwrite, sub_bcarry)),
   "(dec_ctl(inhibit)), "(mem(idle)),
   "(srcdst(inst_rf,inst_df,X,X)), "(enable(res)), "(select(X, X, m)))"
 );

let SUBS_u2_mc = new_definition
('SUBS_u2_mc',
  "SUBS_u2_mc =
let XOR_u2_mc = new_definition
('X7, "(seq_alu_ctl(stop_pcwrite, xor)),
"(dec_ctl(inhibit)), "(mem(idle)),
"(srcdst(inst_rf,inst_df,X,X)), "(enable(res)), "(select(X, X, m)))"
);

let AND_u2_mc = new_definition
('AND_u2_mc',
"AND_u2_mc =
('X7, "(seq_alu_ctl(stop_pcwrite, and)), "(dec_ctl(inhibit)), "(mem(idle)),
"(srcdst(inst_rf,inst_df,X,X)), "(enable(res)), "(select(X, X, m)))"
);

let NOR_u2_mc = new_definition
('NOR_u2_mc',
"NOR_u2_mc =
('X7, "(seq_alu_ctl(stop_pcwrite, nor)), "(dec_ctl(inhibit)), "(mem(idle)),
"(srcdst(inst_rf,inst_df,X,X)), "(enable(res)), "(select(X, X, m)))"
);

%------------------------------------------------------------------------------
% The following were added to pad out fetches so that
% the synchronous interpreter model could be used
%------------------------------------------------------------------------------

let MF3_u6w1_mc = new_definition
('MF3_u6w1_mc',
"MF3_u6w1_mc =
('wait_0, "(seq_alu_ctl(jmp, idle)), "(dec_ctl(inhibit)), "(mem(idle)),
"(srcdst(X,X,X,X)), "(enable(none)), "(select(X, X, X))"
);

let MF3_u1w4_mc = new_definition
('MF3_u1w4_mc',
"MF3_u1w4_mc =
('wait_4, "(seq_alu_ctl(jmp, mthro)), "(dec_ctl(inhibit)), "(mem(idle)),
"(srcdst(X,m_df,X,regM)), "(enable(res)), "(select(X, X, addr))"
);

let MF3_u5w3_mc = new_definition
let WAIT_mc = new_definition
('WAIT_mc',
"WAIT_mc =
\text{(X7, (seq\_alu\.ctl(idle, idle)), (dec\_ctl(inhibit)),
\text{(mem(idle)), (srcdst(X,X,X,X)), (enable(data)), (select(X, m, X)))})"
);

This list must contain the microinstructions that implement the behavior in the definition micro\_inst\_list defined in def\_micro.ml.

let micro\_rom = new_definition
('micro\_rom',
"!n . micro\_rom n =
EL n
[FETCH\_u1\_mc;
 FETCH\_u2\_mc;
 FETCH\_u3\_mc;
 FETCH\_u4\_mc;
 JMP\_reqmc\_mc;
 JMP\_opc\_mc;
 NOOP\_mc;
 SHRS\_u1\_mc;
 SHRB\_u1\_mc;
 SHLB\_u1\_mc;
 AXY\_WRITE\_mc;
 SHLS\_u1\_mc;
 NO\_OVL\_mc;
 NOOP\_mc;
 AXY\_WRITE\_mc;
 SHRS\_u2\_mc;
 NOOP\_mc;
 AXY\_WRITE\_mc;
 SHRB\_u2\_mc;
 NOOP\_mc;
 AXY\_WRITE\_mc;
 SHLB\_u2\_mc;
 NOOP\_mc;
 MFO\_u1\_mc;
 MFI\_u1\_mc;"
READIO_u2_mc;
MF3_u5_mc;
READIO_u4_mc;
NOOP_mc;
READMEM_u2_mc;
CK_VALID_PC_mc;
NOOP_mc;
ADDB_u2_mc;
NOOP_mc;
ADDS_u2_mc;
CK_VALID_PC_mc;
NO_OVL_mc;
NOOP_mc;
SUBB_u2_mc;
NOOP_mc;
SUBS_u2_mc;
CK_VALID_PC_mc;
NO_OVL_mc;
NOOP_mc;
XOR_u2_mc;
NOOP_mc;
AND_u2_mc;
NOOP_mc;
NOR_u2_mc;
NOOP_mc;
WAIT_mc;
WAIT_mc;
WAIT_mc;
WAIT_mc;
MF3_u6_mc;
NOOP_mc;
NOOP_mc;
NOOP_mc;
NOOP_mc;
NOOP_mc;
NOOP_mc;
NOOP_mc;
NOOP_mc;
NOOP_mc;
NOOP_mc;
NOOP_mc;
NOOP_mc;
NOOP_mc;
NOOP_mc;
NOOP_mc;
NOOP_mc;
NOOP_mc;
NOOP_mc;
NOOP_mc;
NOOP_mc;
NOOP_mc;
NOOP_mc;
save_thm('micro_rom_expanded',
SUBS [FETCH_u1_mc; FETCH_u2_mc; FETCH_u3_mc; FETCH_u4_mc; JMP_reqm_mc;
  JMP_opc_mc; NOOP_mc; SHRS_u1_mc; SHRB_u1_mc; SHLS_u1_mc; SHLB_u1_mc;
  SHRS_u2_mc; SHRB_u2_mc; MF0_u1_mc; MF3_u6e1_mc;
  MF1_u1_mc; MF2_u1_mc; MF3_u1_mc; MF3_u2_mc; MF3_u4_mc; MF3_u5_mc;
  MF3_u6_mc; MF2_u3_mc; COMPARE_u1_mc; WRITEMEM_u1_mc; WRITEIO_u1_mc;
  NEG_u1_mc; CALL_u1_mc; READIO_u1_mc; READMEM_u1_mc; ADDB_u1_mc;
  ADDS_u1_mc; SUBB_u1_mc; SUBS_u1_mc; XOR_u1_mc; AND_u1_mc; NOR_u1_mc;
  ANDMBAR_u1_mc; COMPARE_u2_mc; WRITEMEM_u2_mc; WRITEIO_u2_mc;
  NEGATE_u2_mc; CALL_u2_mc; CALL_u3_mc; READIO_u2_mc; READIO_u4_mc;
  ADDB_u2_mc; ADDS_u2_mc; SUBB_u2_mc; SUBS_u2_mc; XOR_u2_mc; AND_u2_mc;
  NOR_u2_mc; MF3_u1w4_mc; MF3_u5w3_mc; WAIT_mc; CK_VALID_PC_mc;
  AXY_WRITE_mc; NO_OVL_mc; SHLB_u2_mc; READMEM_u2_mc]
  micro_rom

);;

close_theory();;
%---------------------------------------------------------------

File: def_micro.ml

Description: Defines the behavioral description of the micro
interpreter level

Modified by Tony Leung to add wait states to memory fetches to patch
up instruction micro cycles.

Modified by ETS to include AXY_WR and CK_VAL_PC microinstructions

---------------------------------------------------------------%

set_search_path (search_path() @ lib_dir_list);

loadf 'abstract';

system '/bin/rm micro_def.th';

new_theory 'micro_def';

map new_parent ['tuple'; 'aux_def'; 'regs_def'; 'aux_thms'];

let rep_ty = abstract_type 'aux_def' 'opcode';

let add_bt7 = new_definition
  ('add_bt7',
    "! x y .
    add_bt7 x y =
    bt7_ival ((bt7_val x) + y)"
  );

let FETCH_addr = "(F,F,F,F,F,F)";

let NOOP_addr = "(F,F,F,F,T,F)";

let SHRS1_addr = "(F,F,F,T,T,F)";

let SHRBl_addr = "(F,F,T,F,F,F,T)";

let SHLBl_addr = "(F,F,T,F,T,F,F)";

let MFO_addr = "(F,F,T,F,F,F,F)";

let MFOl_addr = "(F,T,F,F,F,F,F)";

let MFll_addr = "(F,T,F,F,F,T,F)";

let MF21_addr = "(F,T,F,F,T,F,T)";

let BASE_addr = "(F,T,F,F,T,F)";
let COMPARE1_addr = "(F,T,T,T,F,T,T)";;
let WRITEMEM1_addr = "(F,T,T,T,F,T,T)";;
let WRITEIO1_addr = "(F,T,T,T,T,F,T)";;
let NEG1_addr = "(T,F,F,F,F,F,T)";;
let CALL1_addr = "(T,F,F,F,T,F,F)";;
let READIO1_addr = "(T,F,F,T,F,F,F)";;
let WRITEIO1_addr = "(T,F,F,T,T,F,F)";;
let ADDB1_addr = "(T,F,F,T,T,T,T)";;
let ADDS1_addr = "(T,F,T,F,F,F,T)";;
let SUBB1_addr = "(T,F,T,F,F,T,T)";;
let SUBS1_addr = "(T,F,T,F,F,T,T)";;
let XOR1_addr = "(T,F,T,T,F,T,T)";;
let AND1_addr = "(T,F,T,T,F,T,T)";;
let NOR1_addr = "(T,F,T,T,T,F,T,T)";;
let wait_0_addr = "(T,T,F,F,T,F,T)";;
let wait_1_addr = "(T,T,F,F,F,F,F)";;
let wait_2_addr = "(T,T,F,F,F,T,T)";;
let wait_3_addr = "(T,T,F,F,F,T,F)";;
let wait_4_addr = "(T,T,F,F,F,F,T)";;

%-----------------------------------------------------------------------

Micro instruction 57: ANDMBAR - destreg := r /\ m

%-----------------------------------------------------------------------

let ANDMBAR_ul = new_definition

('ANDMBAR_ul',
  '(rep:rep_ty) (regs:(*wordn)list) (m ins din dout:*wordn) (ram:*memory)
   (b stop ov1:bool) (mar:*address) (res:*wordn) (mpc:btT)
   (reset:bool).

ANDMBAR_ul rep (regs,m,ins,din,dout,ram,b,stop,ov1,mar,res,mpc) (reset) =

  let new_stop = ( (DSF rep ins = (F,T,T)) \/
                   (DSF rep ins = (T,F,F)) \/


180
let randmbar = band rep ((EL (bt2_val(RSF rep ins)) regs), bnot rep m) in
stop => (regs, m, ins, din, dout, ram, b, T, ov1, mar, res, "FETCH_addr") |

(new_stop => regs | update_reg regs (DSF rep ins) b randmbar),
m, ins, din, dout, ram, b, new_stop,
new_stop => ov1 | F, mar,
(new_stop => res | randmbar),
(new_stop => (F, F, F, F, F, F) | add bt7 mpc 1)"

东海

(update_reg regs (DSF rep ins) b randmbar, m, ins, din, dout, ram,
b, new_stop, F, mar, randmbar, add bt7 mpc 1)"

);;

save_thm('ANDMBAR_u1', EXPAND_LET_RULE ANDMBAR_u1);;

%--------------------------------------------------------------------------

Micro instruction 0: get instn from mem[pc].

--------------------------------------------------------------------------

let FETCH_u1 = new_definition
('FETCH_u1_def',
"!(rep:'rep_ty) (regs:(*wordn)list) (m ins din dout:*wordn) (ram:*memory)
(b stop ov1:bool) (mar:*address) (res:*wordn) (mpc:bt7)
(reset:bool).
FETCH_u1 rep (regs, m, ins, din, dout, ram, b, stop, ov1, mar, res, mpc) (reset) =

let paddr = address rep (EL p_reg regs) in
stop => (regs, m, ins, din, dout, ram, b, T, ov1, mar, res, "FETCH_addr") |

(new_stop => regs | fetch rep (ram, paddr), dout, ram, b, F, F,
paddr, m, add bt7 mpc 1)"
);

save_thm('FETCH_u1', EXPAND_LET_RULE FETCH_u1);;

%--------------------------------------------------------------------------

Micro instruction 1: increment p

--------------------------------------------------------------------------

let FETCH_u2 = new_definition
('FETCH_u2_def',
"!(rep:'rep_ty) (regs:(*wordn)list) (m ins din dout:*wordn) (ram:*memory)
(b stop ov1:bool) (mar:*address) (res:*wordn) (mpc:bt7)
(reset:bool).
FETCH_u2 rep (regs, m, ins, din, dout, ram, b, stop, ov1, mar, res, mpc) (reset) =

let newp = add rep ((EL p_reg regs), (wordn rep 1)) in
stop => (regs,m,ins,din,dout,ram,b,T,ovl,mar,res,'FETCH_addr) |
(update_reg regs (F,T,T) b newp, m, ins, din, dout, ram, b,
F, aovfl rep ((EL p_reg regs), (wordn rep 1), newp),
mar, newp, add_bt7 mpc 1) "

save_thm('FETCH_u2',EXPAND_LET_RULE FETCH_u2);;

%-------------------------------------------------------------
Micro instruction 2: check if (p+1) is valid
-------------------------------------------------------------
let FETCH_u3 = new_definition
('FETCH_u3_def',
"!(rep:'rep_ty) (regs:(*wordn)list) (m ins din dout:*wordn) (ram:*memory)
(b stop ovl:bool) (mar:*address) (res:*wordn) (mpc:bt7)
(reset:bool).
FETCH_u3 rep (regs,m,ins,din,dout,ram,b,stop,ovl,mar,res,mpc) (reset) =
let new_stop = "(valid_address rep res) in
stop => (regs,m,ins,din,dout,ram,b,T,ovl,mar,res,'FETCH_addr) |
(regs, m, ins, din, dout, ram, b, new_stop,
(new_stop => ovl | F), mar, (new_stop => res | m),
(new_stop => (F,F,F,F,F,F,F) (add_bt7 mpc 1))))"

);;

save_thm('FETCH_u3',EXPAND_LET_RULE FETCH_u3);;

%-------------------------------------------------------------
Micro instruction 3: read instruction into ins register
-------------------------------------------------------------
let FETCH_u4 = new_definition
('FETCH_u4_def',
"!(rep:'rep_ty) (regs:(*wordn)list) (m ins din dout:*wordn) (ram:*memory)
(b stop ovl:bool) (mar:*address) (res:*wordn) (mpc:bt7)
(reset:bool).
FETCH_u4 rep (regs,m,ins,din,dout,ram,b,stop,ovl,mar,res,mpc) (reset) =
stop => (regs,m,ins,din,dout,ram,b,T,ovl,mar,res,'FETCH_addr) |
(regs. m, din, din, dout, ram, b, F, F, mar, m, add_bt7 mpc 1)"

);;

save_thm('FETCH_u4',EXPAND_LET_RULE FETCH_u4);;

%-------------------------------------------------------------
Micro instruction 4: jmp on reqm
-------------------------------------------------------------
let JMP_reqm = new_definition
('JMP_reqm',
\begin{verbatim}
let ins_dec = (decode rep (opcode rep ins, b)) in
let new_stop =
  (FST ins_dec \/ (((FST(SND ins_dec)) = (F,F,T,T,F)) \/
  (FST(SND ins_dec)) = (F,F,T,T,T))) /\ ((MSF rep ins) = (F,F)) in
stop => (regs,m,ins,din,dout,ram,b,ovl,mar,res,\'FETCH_addr) |
  (regs, m, ins, din, dout, ram, b, new_stop, |
  (new_stop => ovl \ F, mar, |
  (new_stop => res \ m), |
  (new_stop => (F,F,F,F,F,F,F) \ J
SND (SND ins_dec) => add_bit (MFO_addr (bt2_val(MSF rep ins)) |
  add_bit mpc 1)) "

save_thm('JMP_reqm',EXPAND_LET_RULE JMP_reqm);

Micro instruction 5: jmp to (noop+opc)
%Micro instruction 5: jmp to (noop+opc)

let JMP_opc = new_definition
('JMP_opc',
  !(rep:'rep_ty) (regs:(\*wordn)list) (m ins din dout:\*wordn) (ram:*memory) |
  (b stop ovl:bool) (mar:*address) (res:*wordn) (mpc:bt7) |
  (reset:bool).
JMP_opc rep (regs,m,ins,din,dout,ram,b,stop,ovl,mar,res,mpc) (reset) =
let ins_dec = (decode rep (opcode rep ins, b)) in
let new_stop =
  (FST ins_dec \/ (((FST(SND ins_dec)) = (F,F,T,T,F)) \/
  (FST(SND ins_dec)) = (F,F,T,T,T))) /\ ((MSF rep ins) = (F,F)) in
stop => (regs,m,ins,din,dout,ram,b,ovl,mar,res,\'FETCH_addr) |
  (regs, m, ins, din, dout, ram, b, new_stop, |
  (new_stop => ovl \ F, mar, |
  (new_stop => res \ m), |
  (new_stop => (F,F,F,F,F,F,F) \ J
SND (SND ins_dec) => add_bit (NOOP_addr (bt2_val(MSF rep ins)) |
  add_bit mpc 1)) "

save_thm('JMP_opc',EXPAND_LET_RULE JMP_opc);

Micro instruction 6: NOOP - goto fetch
%Micro instruction 6: NOOP - goto fetch

let NOOP = new_definition
('NOOP',
  !(rep:'rep_ty) (regs:(\*wordn)list) (m ins din dout:\*wordn) (ram:*memory) |
  (b stop ovl:bool) (mar:*address) (res:*wordn) (mpc:bt7) |
  (reset:bool).
NOOP rep (regs,m,ins,din,dout,ram,b,stop,ovl,mar,res,mpc) (reset) =
\end{verbatim}
stop => (regs, m, ins, din, dout, ram, b, T, ovl, mar, res, "FETCH_addr") | 
(regs, m, ins, din, dout, ram, b, F, F, mar, m, "FETCH_addr")

);;

save_thm('NOOP', EXPAND_LET_RULE NOOP);

%-----------------------------------------------
Micro instruction 7: SHRS - goto shrsl
%-----------------------------------------------
let SHRS_u = new_definition
('SHRS_u',
"!(rep:'rep_ty) (regs:*wordn)list) (m ins din dout:*wordn) (ram:*memory)
   (b stop ovl:bool) (mar:*address) (res:*wordn) (mpc:bt7)
   (reset:bool).
SHRS_u rep (regs, m, ins, din, dout, ram, b, stop, ovl, mar, res, mpc) (reset) =
stop => (regs, m, ins, din, dout, ram, b, T, ovl, mar, res, "FETCH_addr") |
   (regs, m, ins, din, dout, ram, b, F, F, mar, m, "SHRS1_addr")
);

save_thm('SHRS_u', EXPAND_LET_RULE SHRS_u);

%-----------------------------------------------
Micro instruction 8: SHRB - goto shrbl
%-----------------------------------------------
let SHRB_u = new_definition
('SHRB_u',
"!(rep:'rep_ty) (regs:*wordn)list) (m ins din dout:*wordn) (ram:*memory)
   (b stop ovl:bool) (mar:*address) (res:*wordn) (mpc:bt7)
   (reset:bool).
SHRB_u rep (regs, m, ins, din, dout, ram, b, stop, ovl, mar, res, mpc) (reset) =
stop => (regs, m, ins, din, dout, ram, b, T, ovl, mar, res, "FETCH_addr") |
   (regs, m, ins, din, dout, ram, b, F, F, mar, m, "SHRB1_addr")
);

save_thm('SHRB_u', EXPAND_LET_RULE SHRB_u);

%-----------------------------------------------
Micro instruction 9: SHLB - goto shlbl
%-----------------------------------------------
let SHLB_u = new_definition
('SHLB_u',
"!(rep:'rep_ty) (regs:*wordn)list) (m ins din dout:*wordn) (ram:*memory)
   (b stop ovl:bool) (mar:*address) (res:*wordn) (mpc:bt7)
   (reset:bool).
SHLB_u rep (regs, m, ins, din, dout, ram, b, stop, ovl, mar, res, mpc) (reset) =
stop => (regs, m, ins, din, dout, ram, b, T, ovl, mar, res, "FETCH_addr") |
184
Micro instruction 10: AXY_WRITE - check if dest≠ a,x or y

let AXY_WRITE = new_definition
('AXY_WRITE',
'!(rep:'rep Ty) (regs:(.wordn)list) (m ins din dout:*wordn) (ram:*memory)
(b stop ovl:bool) (mar:*address) (res:*wordn) (mpc:bt7)
(reset:bool).
AXY_WRITE rep (regs,m,ins,din,dout,ram,b,stop,ovl,mar,res,mpc)(reset) =
let new_stop = ( (DSF rep ins = (F,T,T)) \/
(DSF rep ins = (T,F,F)) \/
(DSF rep ins = (T,T,F)) \/
(DSF rep ins = (T,T,T)) ) in
stop => (regs,m,ins,din,dout,ram,b,ovl,mar,res,'FETCH_addr) |
(stop => ovl I F), mar,
(new_stop => res | m),
(new_stop => (F,F,F,F,F,F) i add_biT mpc 1) "
);

save_thm('AXY_WRITE',EXPAND_LET_RULE AXY_WRITE);;

Micro instruction 11: SHLS - destreg := shifted value

let SHLS_uI = new_definition
('SHLS_uI',
'!(rep:'rep Ty) (regs:(.wordn)list) (m ins din dout:*wordn) (ram:*memory)
(b stop ovl:bool) (mar:*address) (res:*wordn) (mpc:bt7)
(reset:bool).
SHLS_uI rep (regs,m,ins,din,dout,ram,b,stop,ovl,mar,res,mpc)(reset) =
let sval = shl rep (EL (bt2_val(RSF rep ins)) regs) in
stop => (regs,m,ins,din,dout,ram,b,ovl,mar,res,'FETCH_addr) |
(update_reg regs (DSF rep ins) b sval,m,ins, din, dout, ram, b,
F, bitn rep (EL (bt2_val(RSF rep ins)) regs), mar, sval,
add bt7 mpc 1)"
);

save_thm('SHLS_uI',EXPAND_LET_RULE SHLS_uI);;
let NO_OVL = new_definition
"!(rep:'rep_ty) (regs:*wordn) (m ins din dout:*wordn) (ram:*memory)
(b stop ov1:bool) (mar:*address) (res:*wordn) (mpc:bt7)
(reset:bool).
NO_OVL rep (regs,m,ins,din,dout,ram,b,stop,ov1,mar,res,mpc)(reset) =
\%
let new_stop = ( ov1 ) in%
new_stop => (regs,m,ins,din,dout,ram,b,T,ov1,mar,res,'FETCH_addr) |
(regs, m, ins, din, dout, ram, b, ov1, ov1, mar,
(ovl => res | m),
(ov1 => (F,F,F,F,F,F,F) | add_hi7 mpc 1)) "
);

save_thm('NO_OVL',EXPAND_LET_RULE NO_OVL);

%---------------------------------------------------------------------
Micro instruction 13: - goto fetch (NOOP)
%---------------------------------------------------------------------

%---------------------------------------------------------------------
Micro instruction 14: AXY_WRITE
%---------------------------------------------------------------------

%---------------------------------------------------------------------
Micro instruction 15: SHRS - destreg := shifted value
%---------------------------------------------------------------------

let SHRS_u2 = new_definition
"!(rep:'rep_ty) (regs:*wordn) (m ins din dout:*wordn) (ram:*memory)
(b stop ov1:bool) (mar:*address) (res:*wordn) (mpc:bt7)
(reset:bool).
SHRS_u2 rep (regs,m,ins,din,dout,ram,b,stop,ov1,mar,res,mpc)(reset) =
\%
stop => (regs,m,ins,din,dout,ram,b,T,ov1,mar,res,'FETCH_addr) |
(update_reg regs (DSF rep ins) b
 (shr rep (EL (bt2_val(RSF rep ins)) regs)),
 m, ins, din, dout, ram, b, F, F,
 mar, (shr rep (EL (bt2_val(RSF rep ins)) regs)). add_hi7 mpc 1)";

save_thm('SHRS_u2',EXPAND_LET_RULE SHRS_u2);
let SHRB_u2 = new_definition
    ('SHRB_u2',
    "!(rep:'rep_ty) (regs:(wordn)list) (m ins din dout:wordn) (ram:*memory)
      (b stop ovl:bool) (mar:*address) (res:wordn) (mpc:bt7)
      (reset:bool).
      SHRB_u2 rep (regs,m,ins,din,dout,ram,b,stop,ovl,mar,res,mpc) (reset) =
      let sval = shrb rep ((EL (bt2_val(RSF rep ins)) regs), b) in
      stop => (regs,m,ins,din,dout,ram,b,T,ovl,mar,res,FETCH_addr) |
      (update_reg regs (DSF rep ins) b sval, m, ins, din, dout, ram,
      bit0 rep (EL (bt2_val(RSF rep ins)) regs), F, F, mar, sval,
      add_bit mpc 1)"
    );;

save_thm('SHRB_u2',EXPAND_LET_RULE SHRB_u2);

%---------------------------------------------------------------------
Micro instruction 16: - goto fetch (NOOP)
---------------------------------------------------------------------%

%---------------------------------------------------------------------
Micro instruction 17: AXY_WRITE
---------------------------------------------------------------------%

%---------------------------------------------------------------------
Micro instruction 18: SHRB - destreg := shifted value
---------------------------------------------------------------------%

let SHLB_u2 = new_definition
    ('SHLB_u2',
    "!(rep:'rep_ty) (regs:(wordn)list) (m ins din dout:wordn) (ram:*memory)
      (b stop ovl:bool) (mar:*address) (res:wordn) (mpc:bt7)
      (reset:bool).
      SHLB_u2 rep (regs,m,ins,din,dout,ram,b,stop,ovl,mar,res,mpc) (reset) =
      let sval = shlb rep ((EL (bt2_val(RSF rep ins)) regs), b) in
      stop => (regs,m,ins,din,dout,ram,b,T,ovl,mar,res,FETCH_addr) |
      (update_reg regs (DSF rep ins) b sval, m, ins, din, dout, ram,
bitn rep (EL (bt2_val(RSF rep ins)) reg), F, F, mar, sval, add_bt7 mpc 1"
)

save_thm('SHLB_u2', EXPAND_LET_RULE SHLB_u2);

Micro instruction 22: - goto fetch (NOOP)

Micro instruction 23: fetch m : MF=0 - goto mf01

let MFO_ul = new_definition
('MFO_uI',
"!(rep:rep_ty) (regs:*wordn)list) (m ins din dout:*wordn) (ram:*memory)
(b stop ovl:bool) (mar:*address) (res:*wordn) (mpc:bt7)
(reset:bool).
MFO_ul rep (regs,m,ins,din,dout,ram,b,stop,ovl,mar,res,mpc) (reset) =
stop => (regs,m,ins,din,dout,ram,b.T,ovl,mar,res,"FETCH_addr") |
(regs,m,ins,din,dout,ram,b,F,F,mar,keyup"MF01_addr")
);

save_thm('MFO_uI', EXPAND_LET_RULE MFO_uI);

Micro instruction 24: fetch m : MF=1 - goto mf11

let MF1_ul = new_definition
('MF1_uI',
"!(rep:rep_ty) (regs:*wordn)list) (m ins din dout:*wordn) (ram:*memory)
(b stop ovl:bool) (mar:*address) (res:*wordn) (mpc:bt7)
(reset:bool).
MF1_ul rep (regs,m,ins,din,dout,ram,b,stop,ovl,mar,res,mpc) (reset) =
stop => (regs,m,ins,din,dout,ram,b.T,ovl,mar,res,"FETCH_addr") |
(regs,m,ins,din,dout,ram,b,F,F,mar,keyup"MF11_addr")
);

save_thm('MF1_uI', EXPAND_LET_RULE MF1_uI);

Micro instruction 25: fetch m : MF=2 - goto mf21

let MF2_ul = new_definition
('MF2_uI',
"!(rep:rep_ty) (regs:*wordn)list) (m ins din dout:*wordn) (ram:*memory)
(b stop ovl:bool) (mar:*address) (res:*wordn) (mpc:bt7)
(reset:bool).
MF2_ul rep (regs,m,ins,din,dout,ram,b,stop,ovl,mar,res,mpc) (reset) =
stop => (regs,m,ins,din,dout,ram,b.T,ovl,mar,res,"FETCH_addr") |
(regs,m,ins,din,dout,ram,b,F,F,mar,keyup"MF21_addr")
);
let MF3_u1 = new_definition
('MF3_u1',
"!{rep:'rep_ty} (regs:('wordn)list) (m ins din dout:('wordn) list) (ram:*memory)
(b stop ovl:bool) (mar:*address) (res:*wordn) (mpc:bt7)
(reset:bool).
MF3_u1 rep (regs,m,ins,din,dout,ram,b,stop,ovl,mar,res,mpc) (reset) =
stop => (regs,m,ins,din,dout,ram,b,T,ovl,mar,res,`FETCH_addr)

(let pad rep (address rep ins), ins, din, dout, ram, b, F, F,
mar, pad rep (address rep ins), add_bt7 mpc 1)"
);

save_thm('MF3_u1',EXPAND_LET_RULE MF3_u1);

%---------------------------------------------------------------
% Micro instruction 27: fetch m : MF=3 - addr := m + y
%---------------------------------------------------------------
let MF3_u2 = new_definition
('MF3_u2',
"!{rep:'rep_ty} (regs:('wordn)list) (m ins din dout:('wordn) list) (ram:*memory)
(b stop ovl:bool) (mar:*address) (res:*wordn) (mpc:bt7)
(reset:bool).
MF3_u2 rep (regs,m,ins,din,dout,ram,b,stop,ovl,mar,res,mpc) (reset) =
let mplusy = add rap ((EL y_reg regs),m) in
stop => (regs,m,ins,din,dout,ram,b,T,ovl,mar,res,`FETCH_addr)

(let pad rep (opcode rep ins, address rep mplusy), din, dout, ram, b, F, aovfl rep ((EL y_reg regs), m, mplusy),
mar, mplusy, add_bt7 mpc 1)"
);

save_thm('MF3_u2',EXPAND_LET_RULE MF3_u2);

%---------------------------------------------------------------
% Micro instruction 28: fetch m : MF=3 - check if addr > 20 bits (FETCH_u3)
%---------------------------------------------------------------

%---------------------------------------------------------------

Micro instruction 29: fetch m : MF=3 - get word from mem(addr)

let MF3_u4 = new_definition
('MF3_u4_def',
"!(rep:'rep_ty) (regs:(_wordn)list) (m ins din dout:*wordn) (ram:*memory)
(b stop ovl:bool) (mar:*address) (res:*wordn) (mpc:bt7)
(reset:bool).
MF3_u4 rep (regs,m,ins,din,dout,ram,b,stop,ovl,mar,res,mpc) (reset) =
stop => (regs,m,ins,din,dout,ram,b,T,ovl,mar,res,'FETCH_addr) |
(regs, m, ins, fetch rep (ram, address rep ins), dout, ram,
b, F, F, address rep ins, m, add bt7 mpc 1)"
);

save_thm('MF3_u4',EXPAND_LET_RULE MF3_u4);

%-------------------------------------------------------------------------------
Micro instruction 30: fetch m : MF=3 - read word into m register

let MF3_u5 = new_definition
('MF3_u5_def',
"!(rep:'rep_ty) (regs:(*wordn)list) (m ins din dout:*wordn) (ram:*memory)
(b stop ovl:bool) (mar:*address) (res:*wordn) (mpc:bt7)
(reset:bool).
MF3_u5 rep (regs,m,ins,din,dout,ram,b,stop,ovl,mar,res,mpc) (reset) =
stop => (regs,m,ins,din,dout,ram,b,T,ovl,mar,res,'FETCH_addr) |
(regs, din, ins, din, dout, ram, b, F, F, mar, m, add bt7 mpc 1)"
);

save_thm('MF3_u5',EXPAND_LET_RULE MF3_u5);

%-------------------------------------------------------------------------------
Micro instruction 31: fetch m : MF=3 - goto base+opc wait 1 cycle

let MF3_u6w1 = new_definition
('MF3_u6w1',
"!(rep:'rep_ty) (regs:(_wordn)list) (m ins din dout:*wordn) (ram:*memory)
(b stop ovl:bool) (mar:*address) (res:*wordn) (mpc:bt7)
(reset:bool).
MF3_u6w1 rep (regs,m,ins,din,dout,ram,b,stop,ovl,mar,res,mpc) (reset) =
stop => (regs,m,ins,din,dout,ram,b,T,ovl,mar,res,'FETCH_addr) |
(regs, m, ins, din, dout, ram, b, F, F, mar, m, "wait 0_addr"
);

save_thm('MF3_u6w1',EXPAND_LET_RULE MF3_u6w1);

%-----------------------------------------------------------------------------------
Micro instruction 32: fetch m : MF=0 - M := addr wait 4 cycles

let MF3_u1w4 = new_definition
('MF3_u1w4',
"!(rep:'rep_ty) (regs:(wordn)list) (m ins din dout:*wordn) (ram:*memory)
(b stop ovl:bool) (mar:*address) (res:*wordn) (mpc:bt7)
(reset:bool).
MF3_u1w4 rep (regs,m,ins,din,dout,ram,b,stop,ovl,mar,res,mpc) (reset) =
  stop => (regs, pad rep (address rep ins), ins, din, dout, ram, b, F, F,
  mar, pad rep (address rep ins), "wait_4_addr")
);

save_thm('MF3_u1w4',EXPAND_LET_RULE MF3_u1w4);

%-----------------------------------------------------------------------------

Micro instruction 33: fetch m : MF=0 - goto base+opc (MF3_u6)

%-----------------------------------------------------------------------------

Micro instruction 34: fetch m : MF=1 - get word from mem(addr) (MF3_u4)

%-----------------------------------------------------------------------------

Micro instruction 35: fetch m : MF=1 - read word into m register wait 3 cycles

let MF3_u5w3 = new_definition
('MF3_u5w3_def',
"!(rep:'rep_ty) (regs:*wordn)list (m ins din dout:*wordn) (ram:*memory)
(b stop ovl:bool) (mar:*address) (res:*wordn) (mpc:bt7)
(reset:bool).
MF3_u5w3 rep (regs,m,ins,din,dout,ram,b,stop,ovl,mar,res,mpc) (reset) =
  stop => (regs, pad rep (address rep ins), ins, din, dout, ram, b, F, F,
  mar, pad rep (address rep ins), "wait_3_addr")
);

save_thm('MF3_u5w3',EXPAND_LET_RULE MF3_u5w3);

%-----------------------------------------------------------------------------

Micro instruction 36: fetch m : MF=1 - goto base+opc (MF3_u6)

%-----------------------------------------------------------------------------

Micro instruction 37: fetch m : MF=2 - M := addr (MF3_u1)

%-----------------------------------------------------------------------------
Micro instruction 38: fetch m : MF=2 - addr := m + x

let MF2_u3 = new_definition

('MF2_u3',

"(rep:*rep_t) (regs:(*wordn)list) (m ins din dout:*wordn) (ram:*memory)
(b stop ovl:bool) (mar:*address) (res:*wordn) (mpc:bt7)
(reset:bool).

MF2_u3 rep (regs,m,ins,din,dout,ram,b,stop,ovl,mar,res,mpc) (reset) =

let mplusx = add rep ((EL x_reg regs),m) in

stop => (regs,m,ins,din,dout,ram,b,ovl,mar,res,'FETCH_addr) |

(regs,m, join rep (opcode rep ins, address rep mplusx), din,

dout, ram, b, F, aovfl rep ((EL x_reg regs), m, mplusx),

mar, mplusx, add_hi7 mpc 1)"

);;

save_thm('MF2_u3',EXPAND.LET_RULE MF2_u3);

---------------------------------------------------------------------

Micro instruction 39: fetch m : MF=2 - check if addr > 20 bits (FETCH_u3)
---------------------------------------------------------------------

---------------------------------------------------------------------

Micro instruction 40: fetch m : MF=2 - get word from mem(addr) (MF3_u4)
---------------------------------------------------------------------

---------------------------------------------------------------------

Micro instruction 41: fetch m : MF=2 - read word into m register (MF3_u5)
---------------------------------------------------------------------

---------------------------------------------------------------------

Micro instruction 42: fetch m : MF=2 - goto base+opc (MF3_u6)
---------------------------------------------------------------------

---------------------------------------------------------------------

Micro instruction 43: COMPARE - goto compare1
---------------------------------------------------------------------

let COMPARE_u1 = new_definition

('COMPARE_u1',

"(rep:*rep_t) (regs:(*wordn)list) (m ins din dout:*wordn) (ram:*memory)
(b stop ovl:bool) (mar:*address) (res:*wordn) (mpc:bt7)
(reset:bool).

COMPARE_u1 rep (regs,m,ins,din,dout,ram,b,stop,ovl,mar,res,mpc) (reset) =

stop => (regs,m,ins,din,dout,ram,b,T,ovl,mar,res,'FETCH_addr) |

(regs, m, ins, din, dout, ram, b, F, F, mar, m, "COMPARE1_addr")"

);;

save_thm('COMPARE_u1',EXPAND.LET_RULE COMPARE_u1);
Micro instruction 44: WRITEMEM - goto writemem1

let WRITEMEM_u1 = new_definition
('WRITEMEM_u1',
'!(rep:'repTy) (regs:(*wordn)list) (m ins din dout:*wordn) (ram:*memory)
(b stop ovl:boolean) (mar:*address) (res:*wordn) (mpc:bt7)
(reset:boolean).
WRITEMEM_u1 rep (regs,m,ins,din,dout,ram,b,stop,ovl,mar,res,mpc) (reset) =
stop =>
(regs, m, ins, din, dout, ram, b, F, F, mar, m, "WRITEMEM1_addr")
);

save_thm('WRITEMEM_u1',EXPAND_LET_RULE WRITEMEM_u1);

Micro instruction 45: WRITEIO - goto writeio1

let WRITEIO_u1 = new_definition
('WRITEIO_u1',
'!(rep:'repTy) (regs:(*wordn)list) (m ins din dout:*wordn) (ram:*memory)
(b stop ovl:boolean) (mar:*address) (res:*wordn) (mpc:bt7)
(reset:boolean).
WRITEIO_u1 rep (regs,m,ins,din,dout,ram,b,stop,ovl,mar,res,mpc) (reset) =
stop =>
(regs, m, ins, din, dout, ram, b, T, ovl, mar, res, "FETCH_addr") |
(regs, m, ins, din, dout, ram, b, F, F, mar, m, "WRITEIO1_addr")
);

save_thm('WRITEIO_u1',EXPAND_LET_RULE WRITEIO_u1);

Micro instruction 46: NEG - goto neg1

let NEG_u1 = new_definition
('NEG_u1',
'!(rep:'repTy) (regs:(*wordn)list) (m ins din dout:*wordn) (ram:*memory)
(b stop ovl:boolean) (mar:*address) (res:*wordn) (mpc:bt7)
(reset:boolean).
NEG_u1 rep (regs,m,ins,din,dout,ram,b,stop,ovl,mar,res,mpc) (reset) =
stop =>
(regs, m, ins, din, dout, ram, b, T, ovl, mar, res, "FETCH_addr") |
(regs, m, ins, din, dout, ram, b, F, F, mar, m, "NEG1_addr")
);

save_thm('NEG_u1',EXPAND_LET_RULE NEG_u1);
Micro instruction 47: CALL - goto call1

let CALL_u1 = new_definition
  ('CALL_u1',
   '! (rep:rep_ty) (regs:(*wordn)list) (m ins din dout:*wordn) (ram:*memory)
   (b stop ovl:bool) (mar:*address) (res:*wordn) (mpc:bt7)
   (reset:bool).
   CALL_u1 rep (regs, m, ins, din, dout, ram, b, stop, ovl, mar, res, mpc) (reset) =
   stop => (regs, m, ins, din, dout, ram, b, T, ovl, mar, res, "FETCH_addr") |
   (regs, m, ins, din, dout, ram, b, F, F, mar, m, "CALL1_addr")
  );

save_thm('CALL_u1', EXPAND_LET_RULE CALL_u1);

Micro instruction 48: READIO - goto readio1

let READIO_u1 = new_definition
  ('READIO_u1',
   '! (rep:rep_ty) (regs:(*wordn)list) (m ins din dout:*wordn) (ram:*memory)
   (b stop ovl:bool) (mar:*address) (res:*wordn) (mpc:bt7)
   (reset:bool).
   READIO_u1 rep (regs, m, ins, din, dout, ram, b, stop, ovl, mar, res, mpc) (reset) =
   stop => (regs, m, ins, din, dout, ram, b, T, ovl, mar, res, "FETCH_addr") |
   (regs, m, ins, din, dout, ram, b, F, F, mar, m, "READIO1_addr")
  );

save_thm('READIO_u1', EXPAND_LET_RULE READIO_u1);

Micro instruction 49: READMEM - goto readmem1

let READMEM_u1 = new_definition
  ('READMEM_u1',
   '! (rep:rep_ty) (regs:(*wordn)list) (m ins din dout:*wordn) (ram:*memory)
   (b stop ovl:bool) (mar:*address) (res:*wordn) (mpc:bt7)
   (reset:bool).
   READMEM_u1 rep (regs, m, ins, din, dout, ram, b, stop, ovl, mar, res, mpc) (reset) =
   stop => (regs, m, ins, din, dout, ram, b, T, ovl, mar, res, "FETCH_addr") |
   (regs, m, ins, din, dout, ram, b, F, F, mar, m, "READMEM1_addr")
  );

save_thm('READMEM_u1', EXPAND_LET_RULE READMEM_u1);
Micro instruction 50: ADDB - goto addbl

let ADDB_ul = new_definition
  ('ADDB_ul',
    "!(rep:'rep_ty)(regs:(wordn)list)(m ins din dout:*wordn)(ram:*memory)
    (b stop ovl:bool)(mar:*address)(res:*wordn)(mpc:bt7)
    (reset:bool).
    ADDB_ul rep (regs,m,ins,din,dout,ram,b,stop,ovl,mar,res,mpc) (reset) =
    stop => (regs,m,ins,din,dout,ram,b,T,ovl,mar,res,'FETCH_addr') |
    (regs, m, ins, din, dout, ram, b, F, F, mar, m, "ADDB1_addr")
  );;

save_thm('ADD_B_uL',EXPAND_LET_RULE ADDB_ul);;

Micro instruction 51: ADDS - goto adds1

let ADDS_ul = new_definition
  ('ADDSUL',
    "!(rep:'rep_ty)(regs:(wordn)list)(m ins din dout:*wordn)(ram:*memory)
    (b stop ovl:bool)(mar:*address)(res:*wordn)(mpc:bt7)
    (reset:bool).
    ADDSUL rep (regs,m,ins,din,dout,ram,b,stop,ovl,mar,res,mpc) (reset) =
    stop => (regs,m,ins,din,dout,ram,b,T,ovl,mar,res,'FETCH_addr') |
    (regs, m, ins, din, dout, ram, b, F, F, mar, m, "ADDS1_addr")
  );;

save_thm('ADDS_UL',EXPAND_LET_RULE ADDS_ul);;

Micro instruction 52: SUBB - goto subbl

let SUBB_ul = new_definition
  ('SUBB_ul',
    "!(rep:'rep_ty)(regs:(wordn)list)(m ins din dout:*wordn)(ram:*memory)
    (b stop ovl:bool)(mar:*address)(res:*wordn)(mpc:bt7)
    (reset:bool).
    SUBB_ul rep (regs,m,ins,din,dout,ram,b,stop,ovl,mar,res,mpc) (reset) =
    stop => (regs,m,ins,din,dout,ram,b,T,ovl,mar,res,'FETCH_addr') |
    (regs, m, ins, din, dout, ram, b, F, F, mar, m, "SUBB1_addr")
  );;

save_thm('SUBB_ul',EXPAND_LET_RULE SUBB_ul);;

Micro instruction 53: SUBS - goto subs1
let SUBS ul = new_definition
('SUBS ul',
"!(rep:'rep_ty) (regs:(*wordn)list) (m ins din dout:*wordn) (ram:*memory)
(b stop ovl:bool) (mar:*address) (res:*wordn) (mpc:bt7)
(reset:bool).
SUBS ul rep (regs,m,ins,din,dout,ram,b,stop,ovl,mar,res,mpc) (reset) =
stop =>
(regs,m,ins,din,dout,ram,b,T,ovl,mar,res,'FETCH_addr) |
(regs, m, ins, din, dout, ram, b, F, F, mar, m, "SUBS1_addr")
);;
save_thm('SUBS ul',EXPAND_LET_RULE SUBS ul);

Micro instruction 54: XOR - goto xor1
-----------------------------------------------%-----------------------------------------------
let XOR ul = new_definition
('XOR ul',
"!(rep:'rep_ty) (regs:(*wordn)list) (m ins din dout:*wordn) (ram:*memory)
(b stop ovl:bool) (mar:*address) (res:*wordn) (mpc:bt7)
(reset:bool).
XOR ul rep (regs,m,ins,din,dout,ram,b,stop,ovl,mar,res,mpc) (reset) =
stop =>
(regs,m,ins,din,dout,ram,b,T,ovl,mar,res,'FETCH_addr) |
(regs, m, ins, din, dout, ram, b, F, F, mar, m, "XOR1_addr")
);
save_thm('XOR ul',EXPAND_LET_RULE XOR ul);

Micro instruction 55: AND - goto and1
-----------------------------------------------%-----------------------------------------------
let AND ul = new_definition
('AND ul',
"!(rep:'rep_ty) (regs:(*wordn)list) (m ins din dout:*wordn) (ram:*memory)
(b stop ovl:bool) (mar:*address) (res:*wordn) (mpc:bt7)
(reset:bool).
AND ul rep (regs,m,ins,din,dout,ram,b,stop,ovl,mar,res,mpc) (reset) =
stop =>
(regs,m,ins,din,dout,ram,b,T,ovl,mar,res,'FETCH_addr) |
(regs, m, ins, din, dout, ram, b, F, F, mar, m, "AND1_addr")
);
save_thm('AND ul',EXPAND_LET_RULE AND ul);

Micro instruction 56: NOR - goto nor1
-----------------------------------------------%
let NOR_u1 = new_definition
  ('NOR_u1',
    "(!rep:'rep_ty) (regs:(*wordn)list) (m ins din dout:*wordn) (ram:*memory)
    (b stop ovl:bool) (mar:*address) (res:*wordn) (mpc:bt7)
    (reset:bool).
    NOR_u1 rep (regs,m,ins,din,dout,ram,b,stop,ovl,mar,res,mpc) (reset) =
      stop => (regs,m,ins,din,dout,ram,b,T,ovl,mar,res,'FETCH_addr') |
      (regs, m, ins, din, dout, ram, b, F, F, mar, m, "NOR1_addr")"
  );;

save_thm('NOR_u1',EXPAND_LET_RULE NOR_u1);

%------------------------------------------------------------------------
% Micro instruction 58: ANDMBAR - goto fetch (NGOP)
%------------------------------------------------------------------------

%------------------------------------------------------------------------
% Micro instruction 59: COMPARE - b := compare(r,m)
%------------------------------------------------------------------------

let COMPARE_u2 = new_definition
  ('COMPARE_u2',
    "(!rep:'rep_ty) (regs:(*wordn)list) (m ins din dout:*wordn) (ram:*memory)
    (b stop ovl:bool) (mar:*address) (res:*wordn) (mpc:bt7)
    (reset:bool).
    COMPARE_u2 rep (regs,m,ins,din,dout,ram,b,stop,ovl,mar,res,mpc) (reset) =
      stop => (regs,m,ins,din,dout,ram,b,T,ovl,mar,res,'FETCH_addr') |
      (regs, m, ins, din, dout, ram,
      bcmp rep ((EL (bt2_val(RSF rep ins)) regs),m,b,FSF rep ins), F,
      F, mar, m, add_bt7 mpc 1)"
  );;

save_thm('COMPARE_u2',EXPAND_LET_RULE COMPARE_u2);

%------------------------------------------------------------------------
% Micro instruction 60: COMPARE - goto fetch (NOOP)
%------------------------------------------------------------------------

%------------------------------------------------------------------------
% Micro instruction 61: WRITEMEM - write r to-address ins[0..19] in memory
%------------------------------------------------------------------------

let WRITEMEM_u2 = new_definition
  ('WRITEMEM_u2',
    "(!rep:'rep_ty) (regs:(*wordn)list) (m ins din dout:*wordn) (ram:*memory)
    (b stop ovl:bool) (mar:*address) (res:*wordn) (mpc:bt7)
    (reset:bool).
    WRITEMEM_u2 rep (regs,m,ins,din,dout,ram,b,stop,ovl,mar,res,mpc) (reset) =
stop => (regs, m, ins, din, dout, ram, b, T, ovl, mar, res, "FETCH_addr") |
  (regs, m, ins, din, EL (bt2_val(RSF rep ins)) regs,
  store rep(ram, address rep ins, EL (bt2_val(RSF rep ins)) regs),
  b, F, F, address rep ins, m, add bt7 mpc 1)"
);;

save_thm('WRITEMEM_u2',EXPAND_LET_RULE WRITEMEM_u2);

Micro instruction 62: WRITEMEM - goto fetch (NOOP)

Micro instruction 63: WRITEIO - write r to address ins[0..19] in io

let WRITEIO_u2 = new_definition
  ('WRITEIO_u2',
   "!(rep:'rep_ty) (regs:(.wordn)list) (m ins din dout:*wordn) (ram:*memory)
   (b stop ovl:bool) (mar:*address) (res:*wordn) (mpc:bt7)
   (reset:bool).
   WRITEIO_u2 rep (regs,m,ins,din,dout,ram,b,stop,ovl,mar,res,mpc) (reset) =
   stop => (regs,m,ins,din,dout,ram,b,T,ovl,mar,res,'FETCH_addr') |
   (regs, m, ins, din, EL (bt2_val(RSF rep ins)) regs,
   storeio rep (ram, address rep ins,
   EL (bt2_val(RSF rep ins)) regs),
   b, F, F, address rep ins, m, add bt7 mpc 1)"
);

save_thm('WRITEIO_u2',EXPAND_LET_RULE WRITEIO_u2);

Micro instruction 64: WRITEIO - goto fetch (NOOP)

Micro instruction 65: AXY_WRITE

Micro instruction 66: NEGATE - destreg := -m

let NEGATE_u2 = new_definition
  ('NEGATE_u2',
   "!(rep:'rep_ty) (regs:(*vordn)list) (m ins din dout:*wordn) (ram:*memory)
   (b stop ovl:bool) (mar:*address) (res:*wordn) (mpc:btT)
   (reset:bool).
   NEGATE_u2 rep (regs,m,ins,din,dout,ram,b,stop,ovl,mar,res,mpc) (reset) =
   stop => (regs,m,ins,din,dout,ram,b,T,ovl,mar,res,'FETCH_addr') |
   (regs, m, ins, din, EL (bt2_val(RSF rep ins)) regs,
   store rep(ram, address rep ins, EL (bt2_val(RSF rep ins)) regs),
   b, F, F, address rep ins, m, add bt7 mpc 1)"
)
stop => (regs,m,ins,din,dout,ram,b,T,ovl,mar,res,\"FETCH_addr\") |
(update_reg regs (DSF rep ins) b (neg rep m), m, ins, din, dout,
ram, b, F, F, mar, (neg rep m), add_bt7 mpc 1)"
);

save_thm('NEGATE_u2',EXPAND_LET_RULE NEGATE_u2);

%------------------------------------------------------------------------
Micro instruction 67: NEGATE - goto fetch (NOOP)
%------------------------------------------------------------------------

%------------------------------------------------------------------------
Micro instruction 68: CALL - y := p
%------------------------------------------------------------------------

let CALL_u2 = new_definition
('CALL_u2',
"!(rep:'rep_ty) (regs:(*wordn)list) (m ins din dout:*wordn) (ram:*memory)
(b stop ovl:bool) (mar:*address) (res:*wordn) (mpc:bt7)
(reset:bool).
CALL_u2 rep (regs,m,ins,din,dout,ram,b,stop,ovl,mar,res,mpc) (reset) =
stop => (regs,m,ins,din,dout,ram,b,T,ovl,mar,res,'FETCH_addr') |
(update_reg regs (F,T,F) b (EL p_reg regs), m, ins, din, dout,
ram, b, F, F, mar, (EL p_reg regs), add_bt7 mpc 1)"
);

save_thm('CALL_u2',EXPAND_LET_RULE CALL_u2);

%------------------------------------------------------------------------
Micro instruction 69: CALL - p := m
%------------------------------------------------------------------------

let CALL_u3 = new_definition
('CALL_u3',
"!(rep:'rep_ty) (regs:(*wordn)list) (m ins din dout:*wordn) (ram:*memory)
(b stop ovl:bool) (mar:*address) (res:*wordn) (mpc:bt7)
(reset:bool).
CALL_u3 rep (regs,m,ins,din,dout,ram,b,stop,ovl,mar,res,mpc) (reset) =
stop => (regs,m,ins,din,dout,ram,b,T,ovl,mar,res,'FETCH_addr') |
(update_reg regs (F,T,T) b m, m, ins, din, dout, ram, b, F, F,
mar, m, add_bt7 mpc 1)"
);

save_thm('CALL_u3',EXPAND_LET_RULE CALL_u3);

%------------------------------------------------------------------------
Micro instruction 70: CALL - check msb 12 bits of res (FETCH_u3)
%------------------------------------------------------------------------
%Micro instruction 71: CALL - goto fetch (NOOP)

%Micro instruction 72: READIO - get word from io(addr)

let READIO_u2 = new_definition
  ('READIO_u2_def',
   "!(rep:'rep_ty) (regs:(wordn)list) (m ins din dout:*wordn) (ram:*memory)
    (b stop ovl:bool) (mar:*address) (res:*wordn) (mpc:bt7)
    (reset:bool).
    READIO_u2 rep (regs,m,ins,din,dout,ram,b,stop,ovl,mar,res,mpc) (reset) =
    stop => (regs,m,ins,din,dout,ram,b,T,ovl,mar,res,'FETCH_addr) |
    (regs, m, ins, fetchio rep (ram, address rep ins), dout, ram, b, F, F, address rep ins, m, add bt7 mpc 1)"
  );

save_thm('READIO_u2',EXPAND_LET_RULE READIO_u2);;

%Micro instruction 73: READIO - read word into m register (MF3_u5)

%Micro instruction 74: READIO - destreg := m

let READIO_u4 = new_definition
  ('READIO_u4_def',
   "!(rep:'rep_ty) (regs:(wordn)list) (m ins din dout:*wordn) (ram:*memory)
    (b stop ovl:bool) (mar:*address) (res:*wordn) (mpc:bt7)
    (reset:bool).
    READIO_u4 rep (regs,m,ins,din,dout,ram,b,stop,ovl,mar,res,mpc) (reset) =
    let new_stop = ( (DSF rep ins = (F,T,T)) \/
      (DSF rep ins = (T,F,F)) \/
      (DSF rep ins = (T,F,T)) \/
      (DSF rep ins = (T,T,F)) ) in
    stop => (regs,m,ins,din,dout,ram,b,T,ovl,mar,res,'FETCH_addr) |
    ( new_stop => regs | update_reg regs (DSF rep ins) b m).
  ")
\begin{verbatim}
m, ins, din, dout, ram, b, new_stop, (new_stop => ovl | F), mar,
(new_stop => res | m), (new_stop => (F,F,F,F,F,F,F) | add_bt7 mpc 1)"

save_thm('READIO_u4',EXPAND_LET_RULE READIO_u4);

%------------------------------------------------------------------------
\% Micro instruction 75: READIO - goto fetch (NOOP)
%------------------------------------------------------------------------

%------------------------------------------------------------------------
\% Micro instruction 76: READMEM - destreg := m (READIO_u4)
%------------------------------------------------------------------------

let READMEM_u2 = new_definition
('READMEM_u2',

"!(rep:rep_ty) (regs:*wordn list) (m ins din dout:*wordn) (ram:*memory)
 (b stop ovl:bool) (mar:*address) (res:*wordn) (mpc:bt7)
 (reset:bool).
READMEM_u2 rep (regs,m,ins,din,dout,ram,b,stop,ovl,mar,res,mpc) (reset) =
 stop => (regs,m,ins,din,dout,ram,b,T,ovl,mar,res,'FETCH_addr) |
 (update_reg regs (DSF rep ins) b m, m, ins, din, dout,
 ram, b, F, F, mar, m, add_bt7 mpc 1)"

save_thm('READMEM_u2',EXPAND_LET_RULE READMEM_u2);

%------------------------------------------------------------------------
\% Micro instruction 77: READMEM - check if dest=p /\ result > 20 bits
% This use to be SHLB_u2
%------------------------------------------------------------------------

let CK_VALID_PC = new_definition
('CK_VALID_PC',

"!(rep:rep_ty) (regs:*wordn list) (m ins din dout:*wordn) (ram:*memory)
 (b stop ovl:bool) (mar:*address) (res:*wordn) (mpc:bt7)
 (reset:bool).
CK_VALID_PC rep (regs,m,ins,din,dout,ram,b,stop,ovl,mar,res,mpc) (reset) =
 let new_stop = (((DSF rep ins = (T,T,F)) /\
 (DSF rep ins = (T,T,T)) ) /\
 "(((DSF rep ins = (T,F,F)) /\ ~b) /\
((DSF rep ins = (T,F,T)) /\ b) ) /\
 ((DSF rep ins = (F,T,T)) /\
 (DSF rep ins = (T,F,F)) /\

\end{verbatim}
(DSF rep ins = (T,F,T))) /
  "(valid_address rep res) in
stop => (regs, m, ins, din, dout, ram, b, T, ovl, mar, res,"FETCH_addr) |
  (regs, m, ins, din, dout, ram, b, new_stop,
   (new_stop => ovl | F), mar,
   (new_stop => res | m),
   (new_stop => (F,F,F,F,F,F) | add_bt7 mpc 1)) "
);;
save_thm('CK_VALID_PC',EXPAND_LET_RULE CK_VALID_PC);

%------------------------------------------------------------------------
Micro instruction 78: READMEM - goto fetch (NOOP)
%------------------------------------------------------------------------
%------------------------------------------------------------------------
Micro instruction 79: ADDB - destreg := r+m; b:=carry
%------------------------------------------------------------------------
let ADDB_u2 = new_definition
  ('ADDB_u2',
   "!(rep:'rep_ty) (regs:*wordn)list) (m ins dout:*wordn) (ram:*memory)
    (b stop ovl:bool) (mar:*address) (res:*wordn) (mpc:bt7)
    (reset:bool).
  ADDB_u2 rep (regs,m,ins,din,dout,ram,b,stop,ovl,mar,res,mpc) (reset) =
let rplusm = (add rep ((EL (bt2_val(RSF rep ins)) regs),m)) in
  let new_stop = ( (DSF rep ins = (F,T,T)) \
    (DSF rep ins = (T,F,T)) \
    (DSF rep ins = (T,F,T)) \
    (DSF rep ins = (T,T,F)) \
    (DSF rep ins = (T,T,T))) in
  stop => (regs, m, ins, din, dout, ram, b, T, ovl, mar, res,"FETCH_addr) |
  ( (new_stop => regs | update_reg regs (DSF rep ins) b rplusm),
    m, ins, din, dout, ram,
    (new_stop => b |
    addp rep((EL (bt2_val(RSF rep ins)) regs),m,rplusm)),
    new_stop,
    (new_stop => ovl |
    aovfl rep ((EL (bt2_val(RSF rep ins)) regs), m, rplusm)),
    mar,
    (new_stop => res | rplusm),
    (new_stop => (F,F,F,F,F,F) | add_bt7 mpc 1)) "
);

save_thm('ADDB_u2',EXPAND_LET_RULE ADDB_u2);;

202
%-----------------------------------------------------------------------
Micro instruction 80: ADDB - goto fetch (NOOP)
%-----------------------------------------------------------------------
%
%-----------------------------------------------------------------------
Micro instruction 81: ADDS - destreg := r+m
%-----------------------------------------------------------------------
%
let ADDS_u2 = new_definition
('ADDS_u2',
"!(rep:'rep_ty) (regs:(*wordn)list) (m ins din dout:*wordn) (ram:*memory)
(b stop ovl:bool) (mar:*address) (res:*wordn) (mpc:bt?)
(reset:bool).
ADDS_u2 rep (regs,m,ins,din,dout,ram,b,stop,ovl,mar,res,mpc) (reset) =
let rplusm = (add rep ((EL (bt2_val(KSF rep ins)) regs),m)) in
stop => (regs,m,ins,din,dout,ram,b,T,ovl,mar,res,'FETCH_addr) |
(update_reg regs (DSF rep ins) b rplusm,m, ins, din, dout, ram, 
b, F, aovfl rep ((EL (bt2_val(RSF rep ins)) regs), m, rplusm),
mar, rplusm, add bt7 mpc )")
);
save_thm('ADDS_u2',EXPAND_LET_RULE ADDS_u2);
%
%-----------------------------------------------------------------------
Micro instn 82: CK_VALID_PC
%-----------------------------------------------------------------------
%
%-----------------------------------------------------------------------
Micro instn 83: NO_OVL
%-----------------------------------------------------------------------
%
%-----------------------------------------------------------------------
Micro instruction 84: ADDS - goto fetch (NOOP)
%-----------------------------------------------------------------------
%
%-----------------------------------------------------------------------
Micro instruction 85: SUBB - destreg := r-m; b:=borrow
%-----------------------------------------------------------------------
%
let SUBB_u2 = new_definition
('SUBB_u2',
"!(rep:'rep_ty) (regs:(*wordn)list) (m ins din dout:*wordn) (ram:*memory)
(b stop ovl:bool) (mar:*address) (res:*wordn) (mpc:bt?)
(reset:bool).
SUBB_u2 rep (regs,m,ins,din,dout,ram,b,stop,ovl,mar,res,mpc) (reset) =
let rminusm = (sub rep ((EL (bt2_val(RSF rep ins)) regs),m)) in
let new_stop = ((DSF rep ins = (F,T,T)) \/
(DSF rep ins = (T,F,F)) \/

203
(DSF rep ins = (T,F,T)) \/
(DSF rep ins = (T,T,F)) \/
(DSF rep ins = (T,T,T)) in
stop => (regs,m,ins,din,dout,ram,b,T,ovl,mar,res,"FETCH_addr) |
  ( new_stop => regs | update_reg regs (DSF rep ins) b rminus ),
  m, ins, din, dout, ram,
  (new_stop => b |
    subp rep((EL (bt2_val(RSF rep ins)) regs),m, rminus)
  , new_stop,
  (new_stop => ovl |
    sovfl rep ((EL (bt2_val(RSF rep ins)) regs), m, rminus)
    , mar,
    (new_stop => res | rminus),
    (new_stop => (F,F,F,F,F,F,F) | add_bt7 mpc 1)"
);

save_thm('SUBB_u2', EXPAND_LET_RULE SUBB_u2);

Micro instruction 86: SUBB - goto fetch (NOP)

Micro instruction 87: SUBS - destreg := r-m

let SUBS_u2 = new_definition
('SUBS_u2',
"!(rep:'rep_ty) (regs:(*wordn)list) (m ins din dout:*wordn) (ram:*memory) 
  (b stop ovl:bool) (mar:*address) (res:*wordn) (mpc:bt7) 
  (reset:bool).
SUBS_u2 rep (regs,m,ins,din,dout,ram,b,stop,ovl,mar,res,mpc) (reset) =
let rminusm = (sub rep ((EL (bt2_val(RSF rep ins)) regs),m)) in
stop => (regs,m,ins,din,dout,ram,b,T,ovl,mar,res,"FETCH_addr) |
  (update_reg regs (DSF rep ins) b rminusm,m,ins,din,dout,ram,
    b, T, sovfl rep ((EL (bt2_val(RSF rep ins)) regs), m, rminusm),
    mar, rminusm, add_bt7 mpc 1)"
);

save_thm('SUBS_u2', EXPAND_LET_RULE SUBS_u2);

Micro instruction 88: CK_VALID_PC

Micro instruction 89: NO_OVL
Micro instruction 90: SUBS - goto fetch (NOOP)

Micro instruction 91: XOR - destreg := r XOR m

let XOR_u2 = new_definition
('XOR_u2',
"!(rep:'rep_ty) (regs:(wordn)list) (m ins din dout:wordn) (ram:*memory)
 (b stop ovl:bool) (mar:*address) (res:wordn) (mpc:btT)
 (reset:bool).
XOR_u2 rep (regs,m,ins,din,dout,ram,b,stop,ovl,mar,res,mpc) (reset) =
let new_stop = ( (DSF rep ins = (F,T,T)) /
 (DSF rep ins = (T,F,T)) /
 (DSF rep ins = (T,F,F)) /
 (DSF rep ins = (T,T,T))) in
let rxorm = (bxor rep ((EL (bt2_val(RSF rep ins)) regs),m)) in
stop => (regs,m,ins,din,dout,ram,b,ovl,mar,res,FETCH_addr) |
 ( (new_stop => regs | update_reg regs (DSF rep ins) b rxorm ),
 m, ins, din, dout, ram, b, new_stop,
 (new_stop => ovl | F), mar,
 (new_stop => res | rxorm),
 (new_stop => (F,F,F,F,F,F | add_bt7 mpc 1) )")
);

save_thm('XOR_u2',EXPAND_LET_RULE XOR_u2);

Micro instruction 92: XOR - goto fetch (NOOP)

Micro instruction 93: AND - destreg := r AND m

let AND_u2 = new_definition
('AND_u2',
"!(rep:'rep_ty) (regs:(wordn)list) (m ins din dout:wordn) (ram:*memory)
 (b stop ovl:bool) (mar:*address) (res:wordn) (mpc:btT)
 (reset:bool).
AND_u2 rep (regs,m,ins,din,dout,ram,b,stop,ovl,mar,res,mpc) (reset) =
let new_stop = ( (DSF rep ins = (F,T,T)) /
 (DSF rep ins = (T,F,F)) /
 (DSF rep ins = (T,T,T)) ) in
let rxorm = (bxor rep ((EL (bt2_val(RSF rep ins)) regs),m)) in
stop => (regs,m,ins,din,dout,ram,b,ovl,mar,res,FETCH_addr) |
 ( (new_stop => regs | update_reg regs (DSF rep ins) b rxorm ),
 m, ins, din, dout, ram, b, new_stop,
 (new_stop => ovl | F), mar,
 (new_stop => res | rxorm),
 (new_stop => (F,F,F,F,F,F | add_bt7 mpc 1) )")
);

save_thm('AND_u2',EXPAND_LET_RULE AND_u2);
(DSF rep ins = (T,F,T)) \/
(DSF rep ins = (T,T,F)) \/
(DSF rep ins = (T,T,T)) in
let randm = (band rep ((EL (bt2_val(RSF rep ins)) regs),m)) in
  stop => (regs,m,ins,din,dout,ram,b,T,ovl,mar,res,"FETCH_addr") |
  (new_stop => regs | update_reg regs (DSF rep ins) b randm),
  m, ins, din, dout, ram, b, new_stop,
  (new_stop => ovl | F), mar,
  (new_stop => res | randm),
  (new_stop => (F,F,F,F,F,F,F) | add_bit mpc 1 ))

);;

save_thm('AND_u2',EXPAND_LET_KULE AND_u2);

%---------------------------------------------------------------
Micro instruction 94: AND - goto fetch (NOOP)
---------------------------------------------------------------%

%---------------------------------------------------------------
Micro instruction 95: NOR - destreg := r NOR m
---------------------------------------------------------------%
let NOR_u2 = new_definition
  (NOR_u2',
   "!(rep:'rep ty) (regs:(wordn)list) (m ins din dout:*wordn) (ram:*memory)
    (b stop ovl:bool) (mar:*address) (res:*wordn) (mpc:btT)
    (reset:bool).
    NOR_u2 rep (regs,m,ins,din,dout,ram,b,stop,ovl,mar,res,mpc) (reset) =
    let new_stop = ( (DSF rep ins = (F,T,T)) \/
      (DSF rep ins = (T,F,T)) \/
      (DSF rep ins = (T,T,F)) \/
      (DSF rep ins = (T,T,T)) ) in
    let rnorm = (bnor rep ((EL (bt2_val(RSF rep ins)) regs),m)) in
    stop => (regs,m,ins,din,dout,ram,b,ovl,mar,res,"FETCH_addr") |
    (new_stop => regs | update_reg regs (DSF rep ins) b rnorm),
    m, ins, din, dout, ram, b, new_stop,
    (new_stop => ovl | F), mar,
    (new_stop => res | rnorm),
    (new_stop => (F,F,F,F,F,F,F) | add_bit mpc 1 ))
  )

);;

save_thm('NOR_u2',EXPAND_LET_RULE NOR_u2);

%---------------------------------------------------------------
Micro instruction 96: NOR - goto fetch (NOOP)

206
Micro instruction 97: wait 4 cycles

let wait_4 = new_definition
  ('wait_4',
   "!(rep:'rep_ty) (regs:(*wordn)list) (m ins din dout:*wordn) (ram:*memory)
   (b stop ovl:boolean) (mar:*address) (res:*wordn) (mpc:bt7)
   (reset:boolean).
   wait_4 rep (regs,m,ins,din,dout,b,stop,ovl,mar,res,mpc) (reset) =
   stop => (regs,m,ins,din,dout,ram,b,T,ovl,mar,res,'FETCH_adcLr')
   (regs, m, ins, din, dout, ram, b, F, F, mar, m, add_bt7 mpc 1)"
 );;

save_thm('wait_4',EXPAND_LET_RULE wait_4);

Micro instruction 98: wait 3 cycles

let wait_3 = new_definition
  ('wait_3',
   "!(rep:'rep_ty) (regs:(*wordn)list) (m ins din dout:*wordn) (ram:*memory)
   (b stop ovl:boolean) (mar:*address) (res:*wordn) (mpc:bt7)
   (reset:boolean).
   wait_3 rep (regs,m,ins,din,dout,b,stop,ovl,mar,res,mpc) (reset) =
   stop => (regs,m,ins,din,dout,ram,b,T,ovl,mar,res,'FETCH_addr')
   (regs, m, ins, din, dout, ram, b, F, F, mar, m, add_bt7 mpc 1)"
 );;

save_thm('wait_3',EXPAND_LET_RULE wait_3);

Micro instruction 99: wait 2 cycles

let wait_2 = new_definition
  ('wait_2',
   "!(rep:'rep_ty) (regs:(*wordn)list) (m ins din dout:*wordn) (ram:*memory)
   (b stop ovl:boolean) (mar:*address) (res:*wordn) (mpc:bt7)
   (reset:boolean).
   wait_2 rep (regs,m,ins,din,dout,b,stop,ovl,mar,res,mpc) (reset) =
   stop => (regs,m,ins,din,dout,ram,b,T,ovl,mar,res,'FETCH_addr')
   (regs, m, ins, din, dout, ram, b, F, F, mar, m, add_bt7 mpc 1)"
 );;

save_thm('wait_2',EXPAND_LET_RULE wait_2);
%-----------------------------------------------------------
Micro instruction 100: wait 1 cycle
-----------------------------------------------------------

let wait_1 = new_definition

('wait_1',
  !(rep:'rep_ty) (regs:(*wordn)list) (m ins din dout:*wordn) (ram:*memory)
  (b stop ovl:bool) (mar:*address) (res:*wordn) (mpc:bt7)
  (reset:bool).
wait_1 rep (regs,m,ins,din dout,ram,b,stop,ovl,mar,res,mpc) (reset) =
  stop => (regs,m,ins,din dout,ram,b,T,ovl,mar,res,"FETCH_addr") |
  (regs, m, ins, din, dout, ram, b, F, F, mar, m, add_bt7 mpc 1 )"
);

save_thm('wait_1',EXPAND_LET_RULE wait_1);

%-----------------------------------------------------------
Micro instruction 101: MF3_u6
-----------------------------------------------------------

let MF3_u6 = new_definition

('MF3_u6',
  !(rep:'rep_ty) (regs:(*wordn)list) (m ins din dout:*wordn) (ram:*memory)
  (b stop ovl:bool) (mar:*address) (res:*wordn) (mpc:bt7)
  (reset:bool).
MF3_u6 rep (regs,m,ins,din dout,ram,b,stop,ovl,mar,res,mpc) (reset) =
  stop => (regs,m,ins,din dout,ram,b,T,ovl,mar,res,"FETCH_addr") |
  (regs, m, ins, din, dout, ram, b, F, F, mar, m,
    add_bt7 "BASE_addr" |
    (bt5_val (FST (SND(decode rep (opcode rep ins, b))))))"
);

save_thm('MF3_u6',EXPAND_LET_RULE MF3_u6);

%-----------------------------------------------------------
Micro instructions 102-127 : NOOP
-----------------------------------------------------------

let micro_state = ":((*wordn)list#*wordn#*wordn#*wordn#*wordn#
  *memory#bool#bool#bool#*address#*wordn#bt7)";

let micro_env = ":bool";;

%-----------------------------------------------------------
The micro_inst_list will be used to instantiate inst_list in
mk_micro.ml.
-----------------------------------------------------------

let micro_inst_list = new_definition

('micro_inst_list',
  "!(rep:'rep_ty)"
);

208
micro_inst_list rep =

[((F,F,F,F,F,F,F),(FETCH_u1 rep)));
((F,F,F,F,F,T,T),(FETCH_u4 rep)));
((F,F,F,F,T,F,T),(JMP_opc rep)));
((F,F,F,F,T,T,F),(NOOP rep)));
((F,F,F,F,T,T,T),(SHRS_u1 rep)));
((F,F,F,T,F,F,F),(SHRB_u1 rep)));
((F,F,F,T,F,F,T),(SHLB_u1 rep)));
((F,F,F,T,F,T,T),(SHLS_u1 rep)));
((F,F,F,T,F,T,F),(NO_OVL rep)));
((F,F,F,T,F,T,T),(NOOP rep)));
((F,F,F,T,T,F,F),(AXY_WRITE rep)));
((F,F,F,T,T,F,T),(SHRS_u2 rep)));
((F,F,T,F,F,F,T),(AXY_WRITE rep)));
((F,F,T,F,F,T,F),(SHRB_u2 rep)));
((F,F,T,F,T,F,F),(NOOP rep)));
((F,F,T,F,T,F,T),(NOOP rep)));
((F,F,T,F,T,F,T),(SHLB_u2 rep)));
((F,F,T,F,T,T,F),(NOOP rep)));
((F,F,T,F,T,T,T),(NOOP rep)));
((F,F,T,F,F,F,F),(MFO_u1 rep)));
((F,F,T,F,F,F,T),(MF1_u1 rep)));
((F,F,T,F,F,T,F),(MF2_u1 rep)));
((F,F,T,F,T,F,F),(MF3_u1 rep)));
((F,F,T,F,T,F,T),(MF3_u2 rep)));
((F,F,T,F,T,F,T),(FETCH_u3 rep)));
((F,F,T,F,T,F,T),(MF3_u4 rep)));
((F,F,T,F,T,T,F),(MF3_u5 rep)));
((F,F,T,F,T,T,T),(MF3_u6 rep)));
((F,T,F,F,F,F,F),(MF3_u4w1 rep)));
((F,T,F,F,F,F,T),(MF3_u4w4 rep)));
((F,T,F,F,F,T,F),(MF3_u6 rep)));
((F,T,F,F,T,F,F),(MF3_u4 rep)));
((F,T,F,F,T,F,T),(MF3_u5w3 rep)));
((F,T,F,F,T,F,T),(MF3_u6 rep)));
((F,T,F,F,T,T,F),(MF3_u1 rep)));
((F,T,F,F,T,T,T),(FETCH_u3 rep)));
((F,T,F,F,T,T,T),(FETCH_u3 rep));
((F,T,F,F,F,T),(MF3_u5 rep));
((F,T,F,F,F,T),(MF3_u6 rep));
((F,T,F,F,T,T),(COMPARE_u1 rep));
((F,T,F,T,F,F),(WRITEEM_u1 rep));
((F,T,F,T,F,T),(WRITEIO_u1 rep));
((F,T,F,T,F,T),(NEG_u1 rep));
((F,T,F,T,T,F),(CALL_u1 rep));
((F,T,F,F,F,F),(READIO_u1 rep));
((F,T,F,F,F,T),(READMEM_u1 rep));
((F,T,F,F,T,F),(ADDB_u1 rep));
((F,T,F,F,T,T),(ADDS_u1 rep));
((F,T,F,T,F,F),(SUBB_u1 rep));
((F,T,F,T,F,T),(SUBS_u1 rep));
((F,T,F,T,F,T),(XOR_u1 rep));
((F,T,F,T,T,F),(AND_u1 rep));
((F,T,F,T,F,F),(NOR_u1 rep));
((F,T,F,T,F,T),(ANDMBAR_u1 rep));
((F,T,T,F,F,T),(COMPARE_u2 rep));
((F,T,T,T,F,F),(NOOP rep));
((F,T,T,T,F,F),(WRITEMEM_u2 rep));
((F,T,T,T,F,F),(NOOP rep));
((F,T,T,T,F,F),(WRITEIO_u2 rep));
((T,F,F,F,F,F),(NOOP rep));
((T,F,F,F,F,F),(AXY_WRITE rep));
((T,F,F,F,F,F),(NEGATE_u2 rep));
((T,F,F,F,F,F),(NOOP rep));
((T,F,F,F,F,F),(CALL_u2 rep));
((T,F,F,F,F,F),(CALL_u3 rep));
((T,F,F,F,F,F),(FETCH_u3 rep));
((T,F,F,F,F,F),(NOOP rep));
((T,F,F,F,F,F),(READIO_u2 rep));
((T,F,F,F,F,F),(MF3_u5 rep));
((T,F,F,F,F,F),(READIO_u4 rep));
((T,F,F,F,F,F),(NOOP rep));
((T,F,F,F,F,F),(READMEM_u2 rep));
((T,F,F,F,F,F),(CK_VALID_PC rep));
((T,F,F,F,F,F),(NOOP rep));
((T,F,F,F,F,F),(ADDB_u2 rep));
((T,F,F,F,F,F),(NOOP rep));
((T,F,F,F,F,F),(ADDS_u2 rep));
((T,F,F,F,F,F),(CK_VALID_PC rep));
((T,T,T,T,F,T),(NOOP rep));
((T,T,T,T,F),(NOOP rep));
((T,T,T,T,T),(NOOP rep))

Select MFC from state. This is used to instantiate gen_I.th.
---------------------------------------------------------------------%
let GetMPC = new_definition
('GetMPC',
"!(regs:(_eordn)list) (m ins din dout:*wordn) (ram:*memory)
(b stop ovl:bool) (mar:*address) (res:*wordn) (mpc:bt7)
(reset:bool).
GetMPC (regs,m,ins,din,dout,ram,b,stop,ovl,mar,res,mpc) (reset) = mpc"
);

%---------------------------------------------------------------------
Give # of phase level cycles for each microinstruction
---------------------------------------------------------------------%
let PhaseCycles = new_definition
('PhaseCycles',
"!key:bt7. PhaseCycles key = 3 "
);

close_theory();

212
File: micro_aux.ml

Description: Defines the micro level interpreter in terms of the definitions in micro_def.th, phase.th, and gen_I.th. Proves the lemmas for each microinstruction and saves them.

Modified by ETS to include AX Y_WRITE and CK_VAL_PC

set_search_path (search_path() @ lib_dir_list);

loadf 'abstract';

% this definition isn't in abstract yet %
let TAC_PROOF : (goal # tactic) -> thm =
  set_fail_prefix 'TAC_PROOF'
  (\(g,tac).
    let new_g = ((fst g) @ theory_obligation_list, snd g) in
    let gl,p = tac new_g in
    if null gl then p[]
    else (message ('Unsolved goals:');
      map print_goal gl;
      print_newline();
      failwith 'unsolved goals'););

system '/bin/rm micro_aux.th';

new_theory 'micro_aux';

% extend_theory 'micro_aux';
%
loadf 'tuple';

map new_parent ['gen_I';'micro_def';'phase';'uinst';'threeval'];
autoload_theory 'ucode_def';
load_definitions 'threeval';
load_theorems 'threeval';

%-----------------------------------------------
From micro_def
let load_micro_inst = (\x. theorem 'micro_def' x);;

let instructions = map load_micro_inst

['FETCH_u1'; 'FETCH_u2'; 'FETCH_u3'; 'FETCH_u4';
'JMP_req'; 'JMP_opc'; 'NOOP'; 'SHRS_u1';
'SHRB_u1'; 'SHLB_u1'; 'AXY_WRITE'; 'SHLS_u1';
'NO_OVL'; 'NOOP'; 'AXY_WRITE'; 'SHRS_u2';
'NOOP'; 'AXY_WRITE'; 'SHRB_u2'; 'NOOP';
'AXY_WRITE'; 'SHLB_u2'; 'NOOP'; 'MF0_u1';
'MF1_u1';
'MF2_u1'; 'MF3_u1'; 'MF3_u2'; 'FETCH_u3';
'MF3_u4'; 'MF3_u5'; 'MF3_u6'; 'MF3_u6I'; 'MF3_u6w1'; 'MF3_u6w4';
'MF3_u6'; 'MF3_u4'; 'MF3_u5'; 'MF2_u3'; 'FETCH_u3'; 'MF3_u4';
'MF3_u5'; 'MF3_u6'; 'COMPARE_u1'; 'WRITEMEM_u1';
'WRITEIO_u1'; 'NEG_u1'; 'CALL_u1'; 'READIO_u1';
'READMEM_u1'; 'ADDB_u1'; 'ADDS_u1'; 'SUBB_u1';
'SUBS_u1'; 'XOR_u1'; 'AND_u1'; 'NOR_u1';
'ANDMBAR_u1'; 'NOOP'; 'COMPARE_u2'; 'NOOP';
'WRITEMEM_u2'; 'NOOP'; 'WRITEIO_u2'; 'NOOP';
'AXY_WRITE'; 'NEGATE_u2'; 'NOOP'; 'CALL_u2';
'CALL_u3'; 'FETCH_u3'; 'NOOP'; 'READIO_u2';
'MF3_u5'; 'READIO_u4'; 'NOOP'; 'READMEM_u2';
'CK_VALID_PC'; 'NOOP'; 'ADDB_u2'; 'NOOP';
'ADDS_u2'; 'CK_VALID_PC'; 'NO_OVL'; 'NOOP';
'SUBB_u2';
'NOOP'; 'SUBS_u2'; 'CK_VALID_PC'; 'NO_OVL';
'NOOP';
'XOR_u2'; 'NOOP'; 'AND_u2'; 'NOOP';
'NOR_u2'; 'NOOP'; 'wait_4'; 'wait_3';
'wait_2'; 'wait_1'; 'MF3_u6'; 'NOOP';
'NOOP'; 'NOOP'; 'NOOP'; 'NOOP';
'NOOP'; 'NOOP'; 'NOOP'; 'NOOP';
'NOOP'; 'NOOP'; 'NOOP'; 'NOOP';
'NOOP'; 'NOOP'; 'NOOP'; 'NOOP';
'NOOP'];;

let micro_inst_list = definition 'micro_def' 'micro_inst_list';;

let GetMPC = definition 'micro_def' 'GetMPC';;

let PhaseCycles = definition 'micro_def' 'PhaseCycles';;

let add_bt7 = definition 'micro_def' 'add_bt7';;
From phase_def

let load_phase_inst = (\x. definition 'phase_def' x);

let phases = map load_phase_ins
    ['phase_one_def'; 'phase_two_def'; 'phase_three_def'];

let PhaseClockBegin = definition 'phase_def' 'PhaseClockBegin';

let Phase_Substate = definition 'phase_def' 'Phase_Substate';

let GetPhaseClock = definition 'phase_def' 'GetPhaseClock';

let Phase_I = theorem 'phase' 'PHASE_I';

let cond3_def = definition 'phase' 'cond3_def';

let cond3_lemma = theorem 'phase' 'cond3_lemma';

let micro_rom_expanded = theorem 'uinst' 'micro_rom_expanded';

let A = definition 'regs_def' 'A';

let X = definition 'regs_def' 'X';

let Y = definition 'regs_def' 'Y';

let P = definition 'regs_def' 'P';

%---------------------------------------------------------

The representation types
%---------------------------------------------------------

let rep_ty = abstract_type 'aux_def' 'opcode';

let I_rep_ty = abstract_type 'gen_I' 'Impl';

let micro_state = "((wordn)list#wordn#wordn#wordn#wordn#
    memory#bool#bool#bool#address#wordn#bt7)#ucode#
    *ordn#wordn#bool#bool#bool";

let micro_env = "bool";

let Phase_state = "((wordn)list #wordn #wordn #wordn #wordn#
    memory #bool #bool #bool #bool #address #wordn #bt7 #ucode #
    (num -> ucode) #
    *wordn # *wordn # bool # bool # bool";

let Phase_env = "bool";

%-------------------------------------

Intermediate theorem needed for rewriting

let ZERO_NEQ_SUC = theorem 'micro_aux' 'ZERO_NEQ_SUC';
%-------------------------------------
let ZERO_NEQ_SUC = prove_thm(  'ZERO_NEQ_SUC',  "!n. ~ (0 = SUC n)",  GEN_TAC THEN REWRITE_TAC[REWRITE_RULE[LESS_0]  (SPECL ['"0"'; '"SUC n"] LESS_NOT_EQ)]) ;;

%---------------------------------------------------------------------
% Define the micro level interpreter in terms of the generic interpreter definition.

let Micro_I_def = definition 'micro_aux' 'Micro_I_def';;

let Micro_I = theorem 'micro_aux' 'Micro_I';;

let Micro_I_IMPL_IMPL_DEF = definition 'micro_aux' 'Micro_I_IMPL_IMPL_DEF';;

--------------------------------------------------

let Micro_I_def = new_definition  ('Micro_I_def _ ,  "! (rep:'rep_ty) (s:time->'micro_state) (e:time->'micro_env) •  Micro_I rep s e  =  INTERP  (micro_inst_list rep,  bt7_val,  (GetMPC:'micro_state -> 'micro_env -> bt7),  (PhaseCycles:bt7->num),  (Phase_Substate:'Phase_state -> 'micro_state),  (I:'Phase_env ->'micro_env),  Phase_I rep,  (GetPhaseClock:'Phase_state -> 'Phase_env -> triple),  PhaseClockBegin, @x:one.F) s e"  ) ;;

let Micro_I = save_thm  ('Micro_I',  ONCE_REWRITE_RULE [GetMPC] (  BETA_RULE (  EXPAND_LET_RULE (instantiate_abstract_definition 'gen_I' INTERP Micro_I_def)))) ;;

let Micro_I_IMPL_IMPL_DEF = new_definition  ('Micro_I_IMPL_IMPL_DEF _ ,  216
Micro_I_IMPL_IMP rep s e =

IMPL_IMP
(micro_inst_list rep,
  (GetMPC: "micro_state -> "micro_env -> bt7),
  (PhaseCycles: bt7 -> num),
  (Phase_Substate: "Phase_state -> "micro_state),
  (I: "Phase_env -> "micro_env),
  Phase_I rep,
  (GetPhaseClock: "Phase_state -> "Phase_env -> triple),
  PhaseClockBegin, @x:one.F) s e
);

let Micro_I_IMPL_IMP =
  let Micro_I_EXT =
    CONV_RULE (TOP_DEPTH_CONV FUN_EQ_CONV) Micro_I_IMPL_IMPL_DEF in
    (REWRITE_RULE [I_THM] (BETA_RULE (EXPAND_LET_RULE (instantiate_abstract_definition 'gen_l' 'IMPL_IMP' Micro_I_EXT))));;

map (delete_cache o fst) (cached_theories());;

%-------------------------------------------------------------------------%
% Some ML function for the inference rules that follow.                    
%-------------------------------------------------------------------------%{a
let rec term_list_el n l =
  let tm_hd x = rand(fst(dest_comb x)) and
    tm_tl x = snd(dest_comb x) in
  if (n = O) then tm_hd I else
    term_list_el (n-1) (tm_tl I) ?
    failwith 'term_list_el';;

%-------------------------------------------------------------------------%
% This is insecure for right now, but it is reasonably simple.              
%-------------------------------------------------------------------------%{a
let EL_CONV tm =
  let ((c,n),l) = ((dest_comb#I) o dest_comb) tm in
  let n_int = term_to_int n in
  mk_thm([], "tm = " (term_list_el n_int l)) ? failwith 'EL_CONV';;
Some other nice conversions

let is_SND_term t =
  if is_comb t then
    fst(dest_const(fst(strip_comb t))) = 'SND'
  else
    false;;

let SND_CONV t =
  if is_SND_term t then
    let op,pr = dest_comb t in
    let op,[t1;t2] = strip_comb pr in
    SPECL [t1;t2] (INST_TYPE [[(type_of t1),"*:"];
                           [(type_of t2),":**"] SND)
  else
    failwith 'SND_CONV';;

let TPLUS3LEMMA = TAC_PROOF
  (\[],
   "'t. (t+3) = (((t + 1) + 1) + 1)"
  )
  STRIP_TAC
  CONV_TAC (TOP_DEPTH_CONV num_CONV) THEN
  REWRITE_TAC[ADD_CLAUSES]

let Phase_I_SPEC =
  PURE_ONCE_REWRITE_RULE [GetPhaseClock] (BETA_RULE
    SPECL ['rep:'rep_ty''
    "'\t.(regs t, mreg t, insreg t, din t, dout t, ram t,
       b t, stop t, ovl t, mar t, res t, mpc t, mir t, micro_rom,
       rlatch t, mlatch t, phi_l t, phi_2 t, phi_3 t)):time->'Phase_state''
    "('t. (reset t)):time->'Phase_env')] Phase_I));

let MK_Phase_I_Inst_LEMMA inst =
  let clk_term =
    ((inst = 1) => "stop t = T" |
     (inst = 2) => "phi_l t = T" |
     (inst = 3) => "phi_2 t = T" |
     "phi_3 t = T") in
  let clk_lemma =
    REWRITE_RULE [] (SUBS [ASSUME clk_term] (SPEC "t" (ASSUME

218
let Phase_I_Inst_list = map MK_Phase_I_Inst_LF
[1;2;3;4];

let Micro_IMPL_IMP_LEMMA = REWRITE_RULE [GetPhaseClock; Phase_Substate; PhaseClockBegin; GetMPC;
PhaseCycles] ( 

BETA_RULE ( 

SPECL ["rep:rep_ty": 

("\t. (regs t, mreg t, insreg t, din t, dout t, ram t,
    b t, stop t, ovl t, mar t, res t, mpq t, mir t, micro_rom,
    rlatch t, mlatch t, ph1 t,
    ph2 t, ph3 t)):time->"Phase_state";

("\t. (reset t)):time->"Phase_env"") 

Micro_I_IMPL_IMP)); 

let MK_IMPL_IMP_GOAL n = 
  let inst = term_list_el n 
    (snd(dest_eq(
      snd(dest_forall(concl micro_inst_list))))) in 
  "!(rep:rep_ty) (regs:time->(_wordn)list)
    (mreg insreg din dout:time->*wordn) (ram:time->.memory)
    (b stop ovl:time->bool) (mar:time->_address) (res:time->_wordn)
    (mpq:time->bt7) (mpc:time->ucode) (rlatch mlatch:time->*wordn)

  (!t. 
    (stop t == "phl t /
    (ph1 t = ~stop t /
    (ph2 t = ~stop t /
    (ph3 t = ~stop t /
    (ph3 t = ~stop t /
    ==> 

  Micro_I_IMPL_IMP rep
    (\t. (regs t, mreg t, insreg t, din t, dout t, ram t,
      b t, stop t, ovl t, mar t, res t, mpq t, mir t, micro_rom,
      rlatch t, mlatch t, ph1 t, ph2 t, ph3 t))

  (\t. (reset t) "inst";; 

  let SPEC_SELECTOR x thm = 
    let inst = snd(dest_eq x) in 
    let (addr,seqalu,dec,mem,srcdst,sel) = 
      (I # (I # (I # (I # (I _ dest_pair))))) 
        (I # (I # (I # (I # dest_pair)))) 
          (I # (I # (I # dest_pair))) 
            (I # (I # dest_pair)) ( 

      (dest_pair inst))))) in 
    let (seq,alu) = (dest_pair seqalu) in 
    let (r,w,io) = 
      (I # dest_pair) ( 
      (dest_pair mem)) in 
    let (mrf, mdf, rfc,rfc) = 

220
let (de,re) = dest_pair en in
let (adrs,ds,ms) = (I # dest_pair) (dest_pair sel) in
SPECL [r;w;io;dec;rfc;dfc;de;re;adrs;ds;mrf;ms;seq;mdf;alu;addr] thm;;

let SPEC_ALL_SELECTORS x = map (SPEC_SELECTOR x)
[Maddr;Seqctl;Aluctl;Dec_ctl;R;W;Io;Mrf;Mdf;Rfc;Dfc; De;Re;Adrs;Ds;Ms];;

map (delete_cache o fst) (cached_theories());;

let IMPL_IMP_TAC n = let inst = term_list_el n
(snd(dest_eq(snd(dest_forall(concl micro_inst_list))))) in
let thm = el (n+1) instructions in
let find_Phase_I_term _m = (let ((x,y),z) = ((dest_comb # I)
(dest_comb tm)) in
(x = "Phase_I (rep:'rep_ty)") ? false in
REPEAT STRIP_TAC
THEN SUBST_TAC [SPEC inst Micro_IMPL_IMP_LEMMA]
THEN REWRITE_TAC [thm]
THEN SUBST_TAC[A;X;Y;P]
THEN STRIP_TAC THEN STRIP_TAC THEN STRIP_TAC
THEN POP_ASSUM(
thm. STRIP_ASSUME_TAC (MULTI_MP
(CONJUNCTS (SPEC_EL ["ph2't':bool"; "ph3 t':bool"]
(REWRITE_RULE[cond3_def] cond3_lemma)) thm))
THEN COND_CASES_TAC
THEN POP_ASSUM_THM_TAC
THENL [
ASSUM_LIST(asl. ASSUME_TAC (REWRITE_RULE([el 1 asl]; (el 2 asl); (el 3 asl)])
(SPEC_ALL (el 6 asl))))
THEN ASSUM_LIST (\x. MAP_EVERY ASSUME_TAC (CONJUNCTS (REWRITE_RULE [PAIR_EQ] (\y. MP y (el 2 x)))) (\y. MP y (el 7 x)) (MATCH_MP (el 1 Phase_I_Inst_list)
(hd (filter (find_Phase_I_term o concl) x))))))))))))}}
THEN ASSUM_LIST (\x. MAP_EVERY ASSUME_TAC ( 
  CONJUNCTS ( 
  REWRITE_RULE [PAIR_EQ] ( 
    (\y. MP y (el 11 x)) ( 
      SPEC "t+1" ( 
        (\y. MP y (el 25 x)) ( 
          MATCH_MP (el 1 Phase_l_Inst_list) 
          (hd (filter (find_Phase_l_term o concl) x)) ))))))))
THEN ASSUM_LIST (\x. MAP_EVERY ASSUME_TAC ( 
  CONJUNCTS ( 
  REWRITE_RULE [PAIR_EQ] ( 
    (\y. MP y (el 11 x)) ( 
      SPEC "(t+l)+l" ( 
        (\y. MP y (el 43 x)) ( 
          MATCH_MP (el 1 Phase_l_Inst_list) 
          (hd (filter (find_Phase_l_term o concl) x)) ))))))))
THEN PURE_ONCE_REWRITE_TAC[TPLUS3LEMMA]
THEN ASM_REWRITE_TAC []

ASSUM_LIST (asl. ASSUME_TAC ( 
  REWRITE_RULE[[el 1 asl); (el 2 asl); (el 3 asl)] 
  (SPEC_ALL (el 6 asl))))
THEN ASSUM_LIST (\x. MAP_EVERY ASSUME_TAC ( 
  CONJUNCTS ( 
  REWRITE_RULE [PAIR_EQ] ( 
    SUBS [CONV_RULE (ONCE_DEPTH_CONV EL_CONV) ( 
      SPEC (int_to_term n) micro_rom_expanded)] ( 
      CONV_RULE (ONCE_DEPTH_CONV bt7_val_CONV) ( 
        SUBS [el 5 x] ( 
          (\y. MP y (el 1 x)) ( 
            SPEC "t:time" ( 
              (\y. MP y (el 7 x)) ( 
                MATCH_MP (el 2 Phase_l_Inst_list) 
                (hd (filter (find_Phase_l_term o concl) x)) )))))))))))
THEN ASSUM_LIST (\x. MAP_EVERY ASSUME_TAC ( 
  CONJUNCTS ( 
  REWRITE_RULE [PAIR_EQ] ( 
    SUBS (SPEC_ALL_SELECTORS (concl (el 6 x))) ( 
      SUBS [el 6 x] ( 
        (\y. MP y (el 2 x)) ( 
          SPEC "t+1" ( 
            (\y. MP y (el 25 x)) ( 
              MATCH_MP (el 3 Phase_l_Inst_list) 
              (hd (filter (find_Phase_l_term o concl) x)) )))))))))))
THEN ASSUM_LIST (\x. if is_eq(concl(el 1 x))

222
then

(let (lhs, rhs) = dest_eq(concl(el 11 x)) in

(ASM_CASES_TAC rhs THENL [

POP_ASSUM (\thm.

(ASSUM_LIST (\x. ASSUME_TAC (REWRITE_RULE x thm))) THEN

ASSUME_TAC thm) THEN

ASSUM_LIST (\x. ASSUME_TAC

(REWRITE_RULE[el 1 x] (el 13 x))) THEN

ASSUM_LIST (\x.

(MAP_EVERY ASSUME_TAC (CONJUNCTS (REWRITE_RULE [PAIR_EQ] (\y. MP y (el 1 x)) (SPEC "(t+1)+1" (\y. MP y (el 46 x)) (MATCH_MP (el 1 Phase_I_Inst_list) (hd (filter

(find(phase I term o concl) x)) )))))))) THEN

PURE_ONCE_REWRITE_TAC[TPLUS3LEMA] THEN

ASM_REWRITE_TAC []

;]

POP_ASSUM (\thm. ASSUME_TAC( REWRITE_RULE[] thm)) THEN

POP_ASSUM (\thm.

(ASSUM_LIST (\x. ASSUME_TAC (REWRITE_RULE x thm))) THEN

ASSUME_TAC thm) THEN

ASSUM_LIST (\x. ASSUME_TAC

(REWRITE_RULE[el 1 x] (el 3 x))) THEN

ASSUM_LIST (\x.

(MAP_EVERY ASSUME_TAC (CONJUNCTS (REWRITE_RULE [PAIR_EQ] (SUBS (SPEC_ALL_SELECTORS (concl (el 9 x))) (SUBS [el 9 x] (\y. MP y (el 1 x)) (SPEC "(t+1)+1" (\y. MP y (el 46 x)) (MATCH_MP (el 1 Phase_I_Inst_list) (hd (filter

(find(phase I term o concl) x)) )))))))) THEN

PURE_ONCE_REWRITE_TAC[TPLUS3LEMA] THEN

if n=4 then

(ASM_REWRITE_TAC[add_b7?] THEN

COND_CASES_TAC

THEN ASSUM_LIST(\asl. ASSUME_TAC (ONCE_REWRITE_RULE [DE_MORGAN_THM] (el 22 asl)))

THEN POP_ASSUM(\thm. REWRITE_TAC[thm]) )

223
else ASM_REWRITE_TAC [add_bt7]
])
else
 (MAP_EVERY ASSUME_TAC (CONJUNCTS
 REWRITE_RULE [PAIR_EQ] (SUBS (SPEC_ALL_SELECTORS (concl (el 6 x))) (SUBS [el 6 x] (\y. MP y (el 1 x)) (SPEC "(t+1)+1" (\y. MP y (el 43 x)) (MATCH_MP (el 4 Phase_I_Inst_list) (hd (filter (find_Phase_I_term o concl) x))))))) THEN PURE_ONCE_REWRITE_TAC[TPLUS3LEMMA] THEN ASM_REWRITE_TAC [] THEN REWRITE_TAC[P; bt2_val; bt3_val] THEN CONV_TAC (TOP_DEPTH_CONV num_CONV) THEN REWRITE_TAC [ZERO_NEQ_SUC; NOT_SUC; INV_SUC_EQ; add_bt7] )));

let PROVE_IMPL_IMP_LEMMA n =
 TAC_PROOF (([3, MK_IMPL_IMP_OOAL n), IMPL_IMP_TAC n));;

let SAVE_INST_LEMMA n =
 let name = (concat 'INST_' (string_of_int n)) in
 save_thm(name,PROVE_IMPL_IMP_LEMMA n);;

map (delete_cache o fst) (cached_theories());;

letrec mk_num_list n m =
 if n = m then [m] else
 (n . (mk_num_list (n+1) m));;

%---------------------------------------------------------------------
The microinstructions be proved and the resulting
theorems will be saved. The theorems for microinstruction n
will be saved under the name INST_n
---------------------------------------------------------------------%

map SAVE_INST_LEMMA (mk_num_list 0 15);;

map (delete_cache o fst) (cached_theories());;

map SAVE_INST_LEMMA (mk_num_list 16 31);;

map (delete_cache o fst) (cached_theories());;
map SAVE_INST_LEMMA (mk_num_list 32 47);;
map (delete_cache o fst) (cached_theories());
map SAVE_INST_LEMMA (mk_num_list 48 63);;
map (delete_cache o fst) (cached_theories());
map SAVE_INST_LEMMA (mk_num_list 64 79);;
map (delete_cache o fst) (cached_theories());
map SAVE_INST_LEMMA (mk_num_list 80 95);;
map (delete_cache o fst) (cached_theories());
map SAVE_INST_LEMMA (mk_num_list 96 111);;
map (delete_cache o fst) (cached_theories());
map SAVE_INST_LEMMA (mk_num_list 112 127);;
map (delete_cache o fst) (cached_theories());
close_theory();
File: mk_micro.ml

Description: Uses the individual correctness lemmas for each micro instruction from micro_aux.th to prove the instruction correctness lemma and complete the Phase to Micro level proof.

set_search_path (search_path() @ lib_dir_list);;

loadf 'abstract';;

% this definition isn't in abstract yet %
let TAC_PROOF : (goal # tactic) -> thm =
  set_fail_prefix 'TAC_PROOF'
  (\(g,tac).
   let new_g = ((fst g) @ theory_obligation_list, snd g) in
   let gl,p = tac ne__g in
   if null gl then p[]
   else (message ('Unsolved goals:');
     map print_goal gl;
     print_newline();
     failwith 'unsolved goals');;

system '/bin/rm micro.th';;

new_theory 'micro';;

map loadf ['tuple'];;

map new_parent ['micro_aux'; 'threeval'];;

load_definitions 'threeval';;

load_theorems 'threeval';;

map (delete_cache o fst) (cached_theories());;

let mk_inst_list n =
  letrec mk_inst_list_aux n m =
    let thm x = (theorem 'micro_aux' (concat 'INST_' (string_of_int x)))
      in
    if n = m then [thm m] else
      ((thm n) . (mk_inst_list_aux (n+1) m)) in
  mk_inst_list_aux 0 n;;

let inst_lemma_list = (mk_inst_list 127);;
let Micro_I_def = definition 'micro_aux' 'Micro_I_def';;

let Micro_I = theorem 'micro_aux' 'Micro_I';;

let Micro_I_IMPL_IMPL_DEF = definition 'micro_aux' 'Micro_I_IMPL_IMPL_DEF';;

let Micro_I_IMPL_IMP =
let Micro_I_EXT =
  CONV_RULE (TOP_DEPTH_CONV FUN_EQ_CONV) Micro_I_IMPL_IMPL_DEF in
  (REWRITE_RULE [I_THM] (BETA_RULE (EXPAND_LET_RULE (instantiate_abstract_definition 'gen_I' 'IMPL_IMP' Micro_I_EXT))));;

let micro_inst_list = definition 'micro_def' 'micro_inst_list';;

let micro_tom = definition 'uinst' 'micro_tom';;

map (delete_cache o fst) (cached_theories());;

let Phase_Substate = definition 'phase_def' 'Phase_Substate';;

let GetPhaseClock = definition 'phase_def' 'GetPhaseClock';;

let PhaseClockBegin = definition 'phase_def' 'PhaseClockBegin';;

let Phase_I = theorem 'phase' 'PHASE_I';;

%---------------------------------------------------------------
  Load abstract type definitions.
%---------------------------------------------------------------
let rep_ty = abstract_type 'aux_def' 'opcode';;

let I_rep_ty = abstract_type 'gen_I' 'Impl';;

%---------------------------------------------------------------
  Define type terms for the state and env.
%---------------------------------------------------------------
let micro_state = ":((*worchn)list#*wordn#*wordn_*wordn#*worcin#,memory#bool$bool#bool#*address#*wordn#bt?)";;

let micro_env = "::bool";;

let phase_state =
  ":(*wordn)list # *wordn # *wordn # *wordn # *wordn # *memory # bool 
   bool # bool # *address # *wordn # bt7 # ucode # (num -+ ucode) # 
   *wordn # *wordn # bool # bool # bool";;
let phase_env = "bool";;

map (delete_cache o fst) (cached_theories());

%-----------------------------------------------------------------------------
% Some ML function for the inference rules that follow.
%-----------------------------------------------------------------------------

letrec term_list_el n l =
  let tm_hd x = rand(fst(dest_comb x)) and
    tm_tl x = snd(dest_comb x) in
  if (n = 0) then tm_hd l else
    term_list_el (n-1) (tm_tl l)) ?
  failwith 'term_list_el';;

%-----------------------------------------------------------------------------
% This is insecure for right now, but it is reasonably simple
%-----------------------------------------------------------------------------

let EL_CONV tm =
  let ((c,n),l) = ((dest_comb#1)o dest_comb) tm in
  let n_int = term_to_int n in
  mk_thm(
    ["tm = (term_list_el n_int l)"],
    "tm = (term_list_el n_int l)") ?
  failwith 'EL_CDNV';;

%-----------------------------------------------------------------------------
% The first obligation of the abstract interpreter theory
%-----------------------------------------------------------------------------

let Micro_I_CORRECT_LEMMA_AUX = TAC_PROOF
  ([],
   "!(rep:'rep_ty) (regs:time->(*wordn)list)
    (mreg insreg din dout:time->*wordn) (ram:time->*memory)
    (b stop ovl:time->bool) (mar:time->*address) (res:time->*wordn)
    (mpc:time->bt7) (mir:time->ucode) (urom:num->ucode)
    (rlatch mlatch:time->*wordn) (phl ph2 ph3:time->bool)
    (reset:time->bool).
    (t.
      (stop t => ph1 t \ ph2 t /\ ph3 t) /
      (ph1 t = ~stop t /\ ph2 t /\ ph3 t) /
      (ph2 t = ~stop t /\ ph1 t /\ ph3 t) /
      (ph3 t = ~stop t /\ ph1 t /\ ph2 t)) =>
    EVERY (Micro_I_IMPL_IMP rep
    (\t.
      (regs t,mreg t,insreg t,din t,dout t,ram t,b t,stop t,
         ovl t,mar t, res t,mir t,urom t,rlatch t,mlatch t,
         ph1 t,ph2 t,ph3 t))
    (\t. reset t)) (micro_inst_list rep")
  )
let Micro_I_LENGTH_LEMMA = TAC_PROOF
(([],
  "! mpc. bt7_val mpc < (LENGTH (micro_inst_list (rep:"rep_ty")))"),
  REPEAT GEN_TAC
  THEN REWRITE_TAC [micro_inst_list;LENGTH]
  THEN STRUCT_CASES_TAC (SPEC "mpc:bt7" SEVEN_TUPLE_VALUE_LEMMA)
  THEN CONV_TAC (DEPTH_CONV bt7_val_CONV)
  THEN CONV_TAC (TOP_DEPTH_CONV num_CONV)
  THEN REWRITE_TAC [LESS_O;LESS_MONO_EQ]
  );;
save_thm('Micro_I_LENGTH_LEMMA',Micro_I_LENGTH_LEMMA);

map (delete_cache o fst) (cached_theories());;

%-------------------------------------------------------------------------%

let Micro_I_ORDER_LEMMA = TAC_PROOF
(([],
  "! mpc. mpc = (FST (EL (bt7_val mpc)
    (micro_inst_list (rep:"rep_ty"))))"),
  REPEAT GEN_TAC
  THEN SUBST_TAC [SPEC "rep:"rep_ty" micro_inst_list]
  THEN STRUCT_CASES_TAC (SPEC "mpc:bt7" SEVEN_TUPLE_VALUE_LEMMA)
  THEN CONV_TAC (ONCE_DEPTH_CONV bt7_val_CONV)
  THEN CONV_TAC (ONCE_DEPTH_CONV EL_CONV)
  THEN REWRITE_TAC []
  );;

%-------------------------------------------------------------------------%

The second obligation of the abstract interpreter theory

%-------------------------------------------------------------------------%

The third obligation of the abstract interpreter theory

%-------------------------------------------------------------------------%
let Micro_I_ORDER_LEMMA = mk_thm([],
    "!mpc:bt7 . mpc = (fst (EL (bt7_val mpc) (micro_inst_list (rep:"rep_ty"))))"
);;
------------------%

save_thm('Micro_I_ORDER_LEMMA',Micro_I_ORDER_LEMMA);

map (delete_cache o fst) (cached_theories());;

let theorem_list = instantiate_abstract_theorems 'gen_I'
    [Micro_I_CORRECT_LEMMA;
    Micro_I_LENGTH_LEMMA;
    Micro_I_ORDER_LEMMA]
    [
    ("rep:"rep_ty",
     "(micro_inst_list (rep:"rep_ty"),
     bt7_val,
     GetMPC:"micro_state->"micro_env->bt7,
     Phase_Substate:"phase_state->"micro_state,
     (I:"phase_env->"micro_env),
     Phase_I rep,
     GetPhaseClock:"phase_state->"phase_env->triple,
     PhaseClockBegin:triple,Qx:one.F")");
    ("e":time->*env'",
    "(\t:time. (reset t):bool)"
    ("s":time->*state'",
    "(\t. (regs t, mreg t, insreg t, din t, dout t, ram t,
    b t, stop t, ovl t, mar t, res t, mpc t,
    mir t, urom, rltch t, mlatch t, phi t,
    ph2 t, ph3 t)):time->"phase_state")
    ]
    'MICRO';

let correct_lemma = snd(hd theorem_list);;

let PHASE_IMPL_MICRO_LEMMA = save_thm
    ('PHASE_IMPL_MICRO_LEMMA',
    BETA_RULE (EXPAND_LET_RULE (ONCE_REWRITE_RULE [Phase_Substate:I_THM;GetPhaseClock;PhaseClockBegin] (BETA_RULE (ONCE_REWRITE_RULE [SYM_RULE Micro_I_def correct_lemma])))));

close_theory();;

230
Appendix F: MICROCODE

<table>
<thead>
<tr>
<th>Cycle</th>
<th>uCode</th>
<th>uLoc</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>fetch_u1</td>
<td>0</td>
<td>fetch macro instruction</td>
</tr>
<tr>
<td>t + 1</td>
<td>fetch_u2</td>
<td>1</td>
<td>increment pc</td>
</tr>
<tr>
<td>t + 2</td>
<td>fetch_u3</td>
<td>2</td>
<td>invalid address (&gt; 20 bits)?</td>
</tr>
<tr>
<td>t + 3</td>
<td>fetch_u4</td>
<td>3</td>
<td>ir ← macro instruction</td>
</tr>
<tr>
<td>t + 4</td>
<td>jmp_reqm</td>
<td>4</td>
<td>require memory?</td>
</tr>
</tbody>
</table>

If no memory fetch is required

<table>
<thead>
<tr>
<th>Cycle</th>
<th>uCode</th>
<th>uLoc</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>t + 5</td>
<td>jmp_opc</td>
<td>5</td>
<td>jump to noop+instruction number</td>
</tr>
</tbody>
</table>

If a memory fetch is required

**Addressing mode: IMMEDIATE**

<table>
<thead>
<tr>
<th>Cycle</th>
<th>uCode</th>
<th>uLoc</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>t + 5</td>
<td>MF0_u1</td>
<td>23</td>
<td>jump to immediate addr mode fetch</td>
</tr>
<tr>
<td>t + 6</td>
<td>MF3_u1w4</td>
<td>32</td>
<td>jump to wait 4</td>
</tr>
<tr>
<td>t + 7</td>
<td>wait4</td>
<td>97</td>
<td>idle</td>
</tr>
<tr>
<td>t + 8</td>
<td>wait3</td>
<td>98</td>
<td>idle</td>
</tr>
<tr>
<td>t + 9</td>
<td>wait2</td>
<td>99</td>
<td>idle</td>
</tr>
<tr>
<td>t + 10</td>
<td>wait1</td>
<td>100</td>
<td>idle</td>
</tr>
<tr>
<td>t + 11</td>
<td>MF3_u6</td>
<td>101</td>
<td>jump to base+instruction number</td>
</tr>
</tbody>
</table>
### Addressing mode: INDIRECT

<table>
<thead>
<tr>
<th>Cycle</th>
<th>uCode</th>
<th>uLoc</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t + 5$</td>
<td>MF1.u1</td>
<td>24</td>
<td>jump to indirect addr mode fetch</td>
</tr>
<tr>
<td>$t + 6$</td>
<td>MF3.u4</td>
<td>34</td>
<td>get word from memory</td>
</tr>
<tr>
<td>$t + 7$</td>
<td>MF3.u5w3</td>
<td>35</td>
<td>read word into m and jump to wait 3</td>
</tr>
<tr>
<td>$t + 8$</td>
<td>wait3</td>
<td>98</td>
<td>idle</td>
</tr>
<tr>
<td>$t + 9$</td>
<td>wait2</td>
<td>99</td>
<td>idle</td>
</tr>
<tr>
<td>$t + 10$</td>
<td>wait1</td>
<td>100</td>
<td>idle</td>
</tr>
<tr>
<td>$t + 11$</td>
<td>MF3.u6</td>
<td>101</td>
<td>jump to base+instruction number</td>
</tr>
</tbody>
</table>

### Addressing mode: INDEXED with x

<table>
<thead>
<tr>
<th>Cycle</th>
<th>uCode</th>
<th>uLoc</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t + 5$</td>
<td>MF2.u1</td>
<td>25</td>
<td>jump to indexed-x addr mode fetch</td>
</tr>
<tr>
<td>$t + 6$</td>
<td>MF3.u1</td>
<td>37</td>
<td>m ← instruction operand</td>
</tr>
<tr>
<td>$t + 7$</td>
<td>MF2.u2</td>
<td>38</td>
<td>addr ← m + x</td>
</tr>
<tr>
<td>$t + 8$</td>
<td>fetch_u3</td>
<td>39</td>
<td>invalid address (&gt; 20 bits)?</td>
</tr>
<tr>
<td>$t + 9$</td>
<td>MF3.u4</td>
<td>40</td>
<td>get word from memory</td>
</tr>
<tr>
<td>$t + 10$</td>
<td>MF3.u5</td>
<td>41</td>
<td>read word into m</td>
</tr>
<tr>
<td>$t + 11$</td>
<td>MF3.u6</td>
<td>42</td>
<td>jump to base+instruction number</td>
</tr>
</tbody>
</table>

### Addressing mode: INDEXED with y

<table>
<thead>
<tr>
<th>Cycle</th>
<th>uCode</th>
<th>uLoc</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t + 5$</td>
<td>MF3.u1</td>
<td>26</td>
<td>m ← instruction operand</td>
</tr>
<tr>
<td>$t + 6$</td>
<td>MF3.u2</td>
<td>27</td>
<td>addr ← m + y</td>
</tr>
<tr>
<td>$t + 7$</td>
<td>fetch_u3</td>
<td>28</td>
<td>invalid address (&gt; 20 bits)?</td>
</tr>
<tr>
<td>$t + 8$</td>
<td>MF3.u4</td>
<td>29</td>
<td>get word from memory</td>
</tr>
<tr>
<td>$t + 9$</td>
<td>MF3.u5</td>
<td>30</td>
<td>read word into m</td>
</tr>
<tr>
<td>$t + 10$</td>
<td>MF3.u6w1</td>
<td>31</td>
<td>jump to wait.0 (MF3.u6)</td>
</tr>
<tr>
<td>$t + 11$</td>
<td>MF3.u6</td>
<td>101</td>
<td>jump to base+instruction number</td>
</tr>
</tbody>
</table>
## Individual Instructions

### Instruction: NOOP # 0

<table>
<thead>
<tr>
<th>Cycle</th>
<th>uCode</th>
<th>uLoc</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>(t+6)</td>
<td>NOOP</td>
<td>6</td>
<td>jump to fetch next macro instruction</td>
</tr>
</tbody>
</table>

### Instruction: SHRS # 1

<table>
<thead>
<tr>
<th>Cycle</th>
<th>uCode</th>
<th>uLoc</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>(t+6)</td>
<td>SHRS.u1</td>
<td>7</td>
<td>jump to shrs code</td>
</tr>
<tr>
<td>(t+7)</td>
<td>AXY_WRITE</td>
<td>14</td>
<td>destination must be register A, X or Y</td>
</tr>
<tr>
<td>(t+8)</td>
<td>SHRS.u2</td>
<td>15</td>
<td>shr operation</td>
</tr>
<tr>
<td>(t+9)</td>
<td>NOOP</td>
<td>16</td>
<td>jump to fetch next macro instruction</td>
</tr>
</tbody>
</table>

### Instruction: SHRΒ # 2

<table>
<thead>
<tr>
<th>Cycle</th>
<th>uCode</th>
<th>uLoc</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>(t+6)</td>
<td>SHRΒ.u1</td>
<td>8</td>
<td>jump to shrb code</td>
</tr>
<tr>
<td>(t+7)</td>
<td>AXY_WRITE</td>
<td>17</td>
<td>destination must be register A, X or Y</td>
</tr>
<tr>
<td>(t+8)</td>
<td>SHRΒ.u2</td>
<td>18</td>
<td>shrb operation</td>
</tr>
<tr>
<td>(t+9)</td>
<td>NOOP</td>
<td>19</td>
<td>jump to fetch next macro instruction</td>
</tr>
</tbody>
</table>

### Instruction: SHLB # 3

<table>
<thead>
<tr>
<th>Cycle</th>
<th>uCode</th>
<th>uLoc</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>(t+6)</td>
<td>SHLB.u1</td>
<td>9</td>
<td>jump to shlb code</td>
</tr>
<tr>
<td>(t+7)</td>
<td>AXY_WRITE</td>
<td>20</td>
<td>destination must be register A, X or Y</td>
</tr>
<tr>
<td>(t+8)</td>
<td>SHLB.u2</td>
<td>21</td>
<td>shlb operation</td>
</tr>
<tr>
<td>(t+9)</td>
<td>NOOP</td>
<td>22</td>
<td>jump to fetch next macro instruction</td>
</tr>
</tbody>
</table>

### Instruction: SHLS # 4

<table>
<thead>
<tr>
<th>Cycle</th>
<th>uCode</th>
<th>uLoc</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>(t+6)</td>
<td>AXY_WRITE</td>
<td>10</td>
<td>destination must be register A, X or Y</td>
</tr>
<tr>
<td>(t+7)</td>
<td>SHLS.u1</td>
<td>11</td>
<td>shls operation</td>
</tr>
<tr>
<td>(t+8)</td>
<td>NO.OVL</td>
<td>12</td>
<td>result must not overflow</td>
</tr>
<tr>
<td>(t+9)</td>
<td>NOOP</td>
<td>13</td>
<td>jump to fetch next macro instruction</td>
</tr>
</tbody>
</table>
### Instruction: CMP # 5

<table>
<thead>
<tr>
<th>Cycle</th>
<th>uCode</th>
<th>uLoc</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>t + 12</td>
<td>COMPARE_u1</td>
<td>43</td>
<td>jump to compare code</td>
</tr>
<tr>
<td>t + 13</td>
<td>bcmp</td>
<td>59</td>
<td>compare operation</td>
</tr>
<tr>
<td>t + 14</td>
<td>NOOP</td>
<td>60</td>
<td>jump to fetch next macro instruction</td>
</tr>
</tbody>
</table>

### Instruction: WRITEM # 6

<table>
<thead>
<tr>
<th>Cycle</th>
<th>uCode</th>
<th>uLoc</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>t + 12</td>
<td>writemem_u1</td>
<td>44</td>
<td>jump to writem code</td>
</tr>
<tr>
<td>t + 13</td>
<td>writemem_u2</td>
<td>61</td>
<td>write r to address</td>
</tr>
<tr>
<td>t + 14</td>
<td>NOOP</td>
<td>62</td>
<td>jump to fetch next macro instruction</td>
</tr>
</tbody>
</table>

### Instruction: WRITEIO # 7

<table>
<thead>
<tr>
<th>Cycle</th>
<th>uCode</th>
<th>uLoc</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>t + 12</td>
<td>writeio_u1</td>
<td>45</td>
<td>jump to writeio code</td>
</tr>
<tr>
<td>t + 13</td>
<td>writeio_u2</td>
<td>63</td>
<td>write r to address</td>
</tr>
<tr>
<td>t + 14</td>
<td>NOOP</td>
<td>64</td>
<td>jump to fetch next macro instruction</td>
</tr>
</tbody>
</table>

### Instruction: NEG # 8

<table>
<thead>
<tr>
<th>Cycle</th>
<th>uCode</th>
<th>uLoc</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>t + 12</td>
<td>NEG_u1</td>
<td>46</td>
<td>jump to neg code</td>
</tr>
<tr>
<td>t + 13</td>
<td>AXY_WRITE</td>
<td>65</td>
<td>destination must be register A,X or Y</td>
</tr>
<tr>
<td>t + 14</td>
<td>NEG_u2</td>
<td>66</td>
<td></td>
</tr>
<tr>
<td>t + 15</td>
<td>NOOP</td>
<td>67</td>
<td>jump to fetch next macro instruction</td>
</tr>
</tbody>
</table>

### Instruction: CALL # 9

<table>
<thead>
<tr>
<th>Cycle</th>
<th>uCode</th>
<th>uLoc</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>t + 12</td>
<td>call_u1</td>
<td>47</td>
<td></td>
</tr>
<tr>
<td>t + 13</td>
<td>call_u2</td>
<td>68</td>
<td></td>
</tr>
<tr>
<td>t + 14</td>
<td>call_u3</td>
<td>69</td>
<td></td>
</tr>
<tr>
<td>t + 15</td>
<td>fetch_u3</td>
<td>70</td>
<td></td>
</tr>
<tr>
<td>t + 16</td>
<td>NOOP</td>
<td>71</td>
<td>jump to fetch next macro instruction</td>
</tr>
</tbody>
</table>
### Instruction: READIO # 10

<table>
<thead>
<tr>
<th>Cycle</th>
<th>uCode</th>
<th>uLoc</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t+12$</td>
<td>readio_u1</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>$t+13$</td>
<td>readio_u2</td>
<td>72</td>
<td></td>
</tr>
<tr>
<td>$t+14$</td>
<td>mf3_u5</td>
<td>73</td>
<td></td>
</tr>
<tr>
<td>$t+15$</td>
<td>readio_u4</td>
<td>74</td>
<td></td>
</tr>
<tr>
<td>$t+16$</td>
<td>NOOP</td>
<td>75</td>
<td>jump to fetch next macro instruction</td>
</tr>
</tbody>
</table>

### Instruction: READM # 11

<table>
<thead>
<tr>
<th>Cycle</th>
<th>uCode</th>
<th>uLoc</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t+12$</td>
<td>readmem_u1</td>
<td>49</td>
<td></td>
</tr>
<tr>
<td>$t+13$</td>
<td>readio_u4</td>
<td>76</td>
<td></td>
</tr>
<tr>
<td>$t+14$</td>
<td>CK_VALID_PC</td>
<td>77</td>
<td></td>
</tr>
<tr>
<td>$t+15$</td>
<td>NOOP</td>
<td>78</td>
<td>jump to fetch next macro instruction</td>
</tr>
</tbody>
</table>

### Instruction: ADDB # 12

<table>
<thead>
<tr>
<th>Cycle</th>
<th>uCode</th>
<th>uLoc</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t+12$</td>
<td>ADDB_u1</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>$t+13$</td>
<td>ADDB_u2</td>
<td>79</td>
<td></td>
</tr>
<tr>
<td>$t+14$</td>
<td>NOOP</td>
<td>80</td>
<td>jump to fetch next macro instruction</td>
</tr>
</tbody>
</table>

### Instruction: ADDS # 13

<table>
<thead>
<tr>
<th>Cycle</th>
<th>uCode</th>
<th>uLoc</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t+12$</td>
<td>ADDS_u1</td>
<td>51</td>
<td></td>
</tr>
<tr>
<td>$t+13$</td>
<td>ADDS_u2</td>
<td>81</td>
<td></td>
</tr>
<tr>
<td>$t+14$</td>
<td>CK_VALID_PC</td>
<td>82</td>
<td></td>
</tr>
<tr>
<td>$t+15$</td>
<td>NO_OVL</td>
<td>83</td>
<td></td>
</tr>
<tr>
<td>$t+16$</td>
<td>NOOP</td>
<td>84</td>
<td>jump to fetch next macro instruction</td>
</tr>
</tbody>
</table>

### Instruction: SUBB # 14

<table>
<thead>
<tr>
<th>Cycle</th>
<th>uCode</th>
<th>uLoc</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t+12$</td>
<td>SUBB_u1</td>
<td>52</td>
<td></td>
</tr>
<tr>
<td>$t+13$</td>
<td>SUBB_u2</td>
<td>85</td>
<td></td>
</tr>
<tr>
<td>$t+14$</td>
<td>NOOP</td>
<td>86</td>
<td>jump to fetch next macro instruction</td>
</tr>
</tbody>
</table>
## Instruction: SUBS # 15

<table>
<thead>
<tr>
<th>Cycle</th>
<th>uCode</th>
<th>uLoc</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>t + 12</td>
<td>SUBS.u1</td>
<td>53</td>
<td></td>
</tr>
<tr>
<td>t + 13</td>
<td>SUBS.u2</td>
<td>87</td>
<td></td>
</tr>
<tr>
<td>t + 14</td>
<td>CK.VALID_PC</td>
<td>88</td>
<td></td>
</tr>
<tr>
<td>t + 15</td>
<td>NO_OVL</td>
<td>89</td>
<td></td>
</tr>
<tr>
<td>t + 16</td>
<td>NOOP</td>
<td>90</td>
<td>jump to fetch next macro instruction</td>
</tr>
</tbody>
</table>

## Instruction: XOR # 16

<table>
<thead>
<tr>
<th>Cycle</th>
<th>uCode</th>
<th>uLoc</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>t + 12</td>
<td>XOR.u1</td>
<td>54</td>
<td></td>
</tr>
<tr>
<td>t + 13</td>
<td>XOR.u2</td>
<td>91</td>
<td></td>
</tr>
</tbody>
</table>
| t + 14 | NOOP    | 92   | jump to fetch next macro instruction |}

## Instruction: AND # 17

<table>
<thead>
<tr>
<th>Cycle</th>
<th>uCode</th>
<th>uLoc</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>t - 12</td>
<td>AND.u1</td>
<td>55</td>
<td></td>
</tr>
<tr>
<td>t - 13</td>
<td>AND.u2</td>
<td>93</td>
<td></td>
</tr>
<tr>
<td>t - 14</td>
<td>NOOP</td>
<td>94</td>
<td>jump to fetch next macro instruction</td>
</tr>
</tbody>
</table>

## Instruction: NOR # 18

<table>
<thead>
<tr>
<th>Cycle</th>
<th>uCode</th>
<th>uLoc</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>t + 12</td>
<td>NOR.u1</td>
<td>56</td>
<td></td>
</tr>
<tr>
<td>t + 13</td>
<td>NOR.u2</td>
<td>95</td>
<td></td>
</tr>
<tr>
<td>t + 14</td>
<td>NOOP</td>
<td>96</td>
<td>jump to fetch next macro instruction</td>
</tr>
</tbody>
</table>

## Instruction: ANDMBAR # 19

<table>
<thead>
<tr>
<th>Cycle</th>
<th>uCode</th>
<th>uLoc</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>t + 12</td>
<td>ANDMBAR.u1</td>
<td>57</td>
<td></td>
</tr>
<tr>
<td>t + 13</td>
<td>NOOP</td>
<td>58</td>
<td>jump to fetch next macro instruction</td>
</tr>
</tbody>
</table>
Appendix G: SAMPLE MACRO TO MICRO LEVEL PROOF

let SHIFT_SYMB_EXEC_TAC =
  NORMAL_SYMB_EXEC 4 T4 THEN DELETE_USTEP_TAC 3
  THEN NORMAL_SYMB_EXEC 5 T5 THEN DELETE_USTEP_TAC 4
  THEN NORMAL_SYMB_EXEC 6 T6 THEN DELETE_USTEP_TAC 5
  THEN JMPOPC_POP_ASSUM_TAC
  THEN NEXT_SYMB_EXEC_TAC 7 THEN DELETE_USTEP_TAC 6
  THEN ASM_CASES_TAC AXY_DSF_CASES
  THEN POP_ASSUM(\thm. ASSUME_TAC (REWRITE_RULE [DE_MORGAN_THM] thm ));;

let SHIFT_BAD_DEST_TAC =
  ASSUM_LIST(\asl. ASSUME_TAC(
    REWRITE_RULE (CONJUNCTS(el 1 asl)) DSF_CASES))
  THEN ASSUM_LIST(\asl.
    IMP_RES_TAC (el (mpc_from_thm (el 3 asl)+1) Micro_Int_Inst_list))
  THEN ASSUM_LIST(\asl. POP_ASSUM(\thm. POP_ASSUM(\thml.
    MAP_EVERY ASSUME_TAC ( (CONJUNCTS (REWRITE_RULE
      ([PAIR_EQ;T8] @ (subtract asl[thm])) thm))))))
  THEN DELETE_USTEP_TAC 7
  % The processor is now stopped due to an addressing exception %
  % specialize and rewrite stop_thm show nothing will change %
  THEN ASSUM_LIST(\asl.
    let curTime = (term_to_int
      (rand(rand(fst( dest_eq(snd(dest_thm(el I asl))))))) ) in
    let endTime =
      (term_to_int (snd(dest_eq(snd(dest_thm (el 17 asl) ))))) in
    ASSUME_TAC( REWRITE_RULE [ (el I asl); (el 5 asl) ; (el 21 asl);
      (sumTHM curTime (endTime-curTime)) ]
    (SPECL [(int_to_term (endTime - curTime )); (t_plus_term curTime]
      stop_thm) ) )
  THEN ASSUM_LIST (\asl. POP_ASSUM(\thm.
    MAP_EVERY ASSUME_TAC ( (CONJUNCTS (REWRITE_RULE
      ([PAIR_EQ] @ (subtract asl[thm])) thm))))
  THEN DELETE_USTEP_TAC 8
  THEN ASM_REWRITE_TAC [PAIR_EQ; update_reg; PAIR_EQ;EL_SET_EL];;

let SHIFT_GOOD_DEST_TACl =
  ASSUM_LIST(\asl. ASSUME_TAC( REWRITE_RULE([el 1 asl]) AXY_IMPI ))
  THEN ASSUM_LIST(\asl.
    IMP_RES_TAC (el (mpc_from_thm (el 3 asl)+1) Micro_Int_Inst_list))
  THEN ASSUM_LIST (\asl. POP_ASSUM(\thm. POP_ASSUM(\thml.
    MAP_EVERY ASSUME_TAC ( (CONJUNCTS (REWRITE_RULE
      ([PAIR_EQ;T8] @ (subtract asl[thm])) thm))))))
THEN NORMAL_POP_ASSUM_TAC
THEN DELETE_USTEP_TAC 7
THEN NEXT_SYMB_EXEC_TAC 9 THEN DELETE_USTEP_TAC 8
THEN NEXT_SYMB_EXEC_TAC 10 THEN DELETE_USTEP_TAC 9
;;

let SHIFT_GOOD_DEST_TAC2 =
  ASSUM_LIST(\asl. DISJ_CASES_TAC (el 14 asl) )
  THENL
  [
    EXPAND_REG_TAC
    THEN ASM_REWRITE_TAC [PAIR_EQ;EL_SET_EL]
    THEN INDEPENDENCE_TAC INDEP_A_UPDATE1 .
    ;
    POP_ASSUM(\thm. DISJ_CASES_TAC thm)
    THEN EXPAND_REG_TAC
    THEN ASM_REWRITE_TAC [PAIR_EQ;EL_SET_EL]
    THENL
      [ INDEPENDENCE_TAC INDEP_X_UPDATEI ;
        INDEPENDENCE_TAC INDEP_Y_UPDATEI ]
  ]
  THEN ASM_REWRITE_TAC [] ;;

let SHIFTB_GOOD_DEST_TAC =
  ASSUM_LIST(\asl. DISJ_CASES_TAC (el 14 asl) )
  THENL
  [
    EXPAND_REG_TAC
    THEN EXPAND_B_TAC
    THEN ASM_REWRITE_TAC [PAIR_EQ;EL_SET_EL]
    THEN INDEPENDENCE_TAC INDEP_A_UPDATE1 .
    ;
    POP_ASSUM(\thm. DISJ_CASES_TAC thm)
    THEN EXPAND_REG_TAC
    THEN EXPAND_B_TAC
    THEN ASM_REWRITE_TAC [PAIR_EQ;EL_SET_EL]
    THENL
      [ INDEPENDENCE_TAC INDEP_X_UPDATE1;
        INDEPENDENCE_TAC INDEP_Y_UPDATE1
      ]
  ]
  THEN ASM_REWRITE_TAC [] ;;
max_print_depth 9;;
set_flag('print_asl_list',true);
set_flag('print_asl_list',false);
asl_print_list := [1];;
asl_print_list := [3;2;1];;
p 1;;
set_flag('print_asl',false);
set_flag('print_gl',false);
set_flag('print_gl',true);
set_flag('show_types',false);
fst(dest_eq(snd(dest_thm(SPEC1_EL_COND_THM))));;
loadf '/usr/csgrad/schubert/bin/init';;

--------------------------------------------------------------------------------
max_print_depth 9;;
system '/bin/rm new_shift.th';;
new_theory 'new_shift';;
loadf 'goals.ml';;
loadf 'stack.ml';;
set_search_path (search_path() Q [ ./%heories/ ];);
loadf '../vinst/inst_aux.ml';;
loadf '../vinst/mk_mac.ml';;
loadf '../vinst/new_inst_aux.ml';;
loadf '../vinst/new_shift_aux.ml';;
map (delete_cache o fst) (cached_theories());;

--------------------------------------------------------------------------------
from oxalis:
Run time: 2719.9s
Run time: 880.2s
Run time: 979.8s
Run time: 1177.5s

239
Run time: 14550.3s (combined steps)

first time on chugjug:
Run time: 1838.6s
Run time: 632.8s
Run time: 707.6s
Run time: 952.1s
Run time: 786.5s
missing last step

Second time with improved last step:
Run time: 1997.1s Intermediate theorems generated: 73834
Run time: 652.6s Intermediate theorems generated: 20909
Run time: 748.7s Intermediate theorems generated: 28525
Run time: 1011.2s Intermediate theorems generated: 26480
Run time: 804.8s Intermediate theorems generated: 29003
Run time: 4029.7s Intermediate theorems generated: 115876

9244.1s/154.07m/2.6h thms generated 294627 32thms/sec

%------------------------------------------------------------

PROVE SHR instruction
On American:

Run time: 915.7s Intermediate theorems generated: 67117
Run time: 93.9s Intermediate theorems generated: 3905
Run time: 395.8s Intermediate theorems generated: 28853
Run time: 379.9s Intermediate theorems generated: 23540
Run time: 419.4s Intermediate theorems generated: 29521
Run time: 1528.5s Intermediate theorems generated: 95274

Run time: 3337.4s/55.6m thms generated: 248210 74thms/sec
Run time: 3259.9s Intermediate theorems generated: 248202

%------------------------------------------------------------

map (delete_cache o fst) (cached_theories());
g(MK_INST_CORRECT_GOAL 1);
e(FETCH_INST_TAC 1
  THEN REWRITE_TAC[write_reg_expanded;load_r_expanded]
  THEN SHIFT_SYMB_EXEC1_TAC
  THENL
    [ SHIFT_BAD_DEST_TAC
    ;SHIFT_GOOD_DEST_TAC1
    THEN SHIFT_GOOD_DEST_TAC2
    ]


240
%---------------------------------------------

PROVE SHRB instruction

On American:
Run time: 979.2s Intermediate theorems generated: 69555
Run time: 98.4s Intermediate theorems generated: 5828
Run time: 377.9s Intermediate theorems generated: 28853
Run time: 453.2s Intermediate theorems generated: 31757
Run time: 1840.6s Intermediate theorems generated: 114624
Run time: 3749.3s/62.5m thms generated 250617 67thms/sec

%---------------------------------------------

map (delete_cache o fst) (cached_theories());;

\( \text{g}( \text{MK\_INST\_CORRECT\_GOAL} \ 2) ;; \)

\( \text{e}( \text{FETCH\_INST\_TAC} \ 2 \)
\( \text{THEN\ REWRITE\_TAC}[\text{write\_reg\_expanded};\text{load\_r\_expanded}] \)
\( \text{THEN\ SHIFT\_SYMB\_EXEC\_TAC} \)
\( \text{THENL} \)
\( \text{[ \text{SHIFT\_BAD\_DEST\_TAC} \}
\( \text{; \text{SHIFT\_GOOD\_DEST\_TAC} }\)
\( \text{THEN \text{SHIFTB\_GOOD\_DEST\_TAC} }\)
\)

%---------------------------------------------

PROVE SHLB instruction

Run time: 1453.2s Intermediate theorems generated: 103886
Run time: 406.9s Intermediate theorems generated: 25436
Run time: 2253.6s Intermediate theorems generated: 146380
Run time: 4113.7/68.7m thms generated 275702 67thms/sec

%---------------------------------------------

map (delete_cache o fst) (cached_theories());;

\( \text{g}( \text{MK\_INST\_CORRECT\_GOAL} \ 3) ;; \)

\( \text{e}( \text{FETCH\_INST\_TAC} \ 3 \)
\( \text{THEN\ REWRITE\_TAC}[\text{write\_reg\_expanded};\text{load\_r\_expanded}] \)
\( \text{THEN\ SHIFT\_SYMB\_EXEC\_TAC} \)
\( \text{THENL} \)
\( \text{[ \text{SHIFT\_BAD\_DEST\_TAC} \)
\( \text{; \text{SHIFT\_GOOD\_DEST\_TAC} }\)
\( \text{THEN \text{SHIFTB\_GOOD\_DEST\_TAC} }\)
% PROVE SHL instruction
The microcode for this instruction is different than the other
shift instructions and so, requires specialization of the tactics

Run time: 1484.8s  Intermediate theorems generated: 106432
Run time: 457.5s   Intermediate theorems generated: 28460
Run time: 568.1s   Intermediate theorems generated: 40518

Attempt 2:
Run time: 1205.8s  Intermediate theorems generated: 98349
Run time: 374.1s   Intermediate theorems generated: 24716
Run time: 550.9s   Intermediate theorems generated: 40518
Run time: 1893.9s  Intermediate theorems generated: 130991
Run time: 24.2s    Intermediate theorems generated: 706
Run time: 78.3s    Intermediate theorems generated: 838
Run time: 1972.3s  Intermediate theorems generated: 136632

map (delete_cache o fst) (cached_theories());
g(MK_INST_CORRECT_GOAL 4 );;
e(FETCH_INST_TAC 4
    THEN REWRITE_TAC[write_reg_expanded;load_r_expanded]
    THEN NORMAL_SYMB_EXEC 4 T4 THEN DELETE_USTEP_TAC 3
    THEN NORMAL_SYMB_EXEC 5 T5 THEN DELETE_USTEP_TAC 4
    THEN NORMAL_SYMB_EXEC 6 T6 THEN DELETE_USTEP_TAC 5
    THEN JMPPOPC_POP_ASSUM_TAC
    THEN ASM_CASES_TAC AXY_DSF_CASES
    THEN POP_ASSUM(\thm. ASSUME_TAC (REWRITE_RULE [DE_MORGAN_THM] thm ))
) ;;

% variation on SHIFT_BAD_DEST_TAC %
e( ASSUM_LIST(\asl. ASSUME_TAC( 
    REWRITE_RULE (CONJUNCTS(el 1 asl)) DSF_CASES))
    THEN ASSUM_LIST(\asl. IMP_RES_TAC (el (mpc_from_thm (el 3 asl)+1) Micro_Int_Inst_list))
    THEN ASSUM_LIST (\asl. POP_ASSUM(\thm. POP_ASSUM(\thm1. 
        MAP_EVERY ASSUME_TAC (\asl. ASSERT_TAC (REWRITE_RULE ([PAIR_EQ;T7] \ (subtract asl[thm]) thm )))))
    THEN DELETE_USTEP_TAC 6

242
% The processor is now stopped due to an addressing exception %
% specialize and rewrite stop_thm show nothing will change %
THEN ASSUME_LIST(\asl.

let curTime = (term_to_int
(rnd(rnd(fst( dest_eq(snd(dest_thm(el 1 asl))))))) ) in
let endTime =
(term_to_int (snd(dest_eq(snd(dest_thm (el 17 asl ))))) ) in
ASSUME_TAC REWRITE_RULE [(el 1 asl); (el 5 asl); (el 21 asl);
(sumTHM curTime (endTime-cu/-Time))]
(SPECL [(int_to_term (endTime - curTime )); (t_plus_term curTime)]
stop_thm)
)
THEN ASSUME_LIST(\asl. POP_ASSUM(\thm.

MAP_EVERY ASSUME_TAC ( (CONJUNCTS (REWRITE_RULE
([PAIR_EQ] @ (subtract asl[thm])) thm )))
)
THEN DELETE_USTEP_TAC 7
THEN % rewrite with address case %
ASSUME_LIST(\asl. REWRITE_TAC (CONJUNCTS (el 15 asl)))
THEN ASM_REWRITE_TAC [PAIR_EQ]
THEN REWRITE_TAC [ELP_SET_ELP]
);

% modification to SHIFT_GOOD_DEST_TAC1 %
e( ASSUME_LIST(\asl. ASSUME_TAC( REWRITE_RULE[(el 1 asl]) AXY_IMP1 ))

THEN ASSUME_LIST(\asl.

IMP_RES_TAC (el (mpc_from_thm (el 3 asl)+1) Micro_Int_Inst_list))
THEN ASSUME_LIST (\asl. POP_ASSUM(\thm. POP_ASSUM(\thm1.

MAP_EVERY ASSUME_TAC ( (CONJUNCTS (REWRITE_RULE
([PAIR_EQ;TT] @ (subtract asl[thm])) thm )))))
)
THEN NORMAL_POP_ASSUM_TAC
THEN DELETE_USTEP_TAC 6
THEN NEXT_SYMB_EXEC_TAC 8 THEN DELETE_USTEP_TAC 7
THEN NEXT_SYMB_EXEC_TAC 9 THEN DELETE_USTEP_TAC 8
THEN ASM_CASES_TAC
"((bitn (rep:'rep_ty)(EL (bt2_val
(RSF rep(fetch rep(ram (t:num),address rep(EL p_reg(reg t))))))
(update_reg(reg t)(F,T,T)(b t)(add rep(EL p_reg(reg t),wordn rep 1)))))
);

%------ overflow case ---------%
e( POP_ASSUM(\thm1. POP_ASSUM(\thm2.

(ASSUME_TAC thm1 )

THEN ASSUME_TAC( REWRITE_RULE [thm1] thm2) ))
THEN ASSUME_LIST(\asl.

let curTime = (term_to_int


243
(rand(rand(fst( dest_eq(snd(dest_thm(el 1 asl)))))) in
  let endTime =
      (term_to_int (snd(dest_eq(snd(dest_thm (el 18 asl) ))))) in
  ASSUME_TAC( REWRITE_RULE([(el 1 asl);(el 6 asl);(el 2 asl);(el 22 asl);
                             (sumTHM curTime (endTime-curTime)) ]
  (SPECL [(int_to_term (endTime - curTime )); (t_plus_term curTime)]
       stop_thm )))
  THEN ASSUM_LIST (asl. POP_ASSUM(\thm.
    MAP_EVERY ASSUME_TAC ( (CONJUNCTS (REWRITE_RULE
      ([PAIR_EQ] @ (subtract asl[thm])) tb_m) )))
  THEN DELETE_USTEP_TAC 9
  % from SHIFT_GOOD_DEST_TAC2  
  THEN ASSUM_LIST(\asl. DISJ_CASES_TAC (el 15 asl) )
  THENL [ EXPAND_REG_TAC
  THEN EXPAND_COND_TAC 15  % rewrite bitn term %
  THEN ASM_REWRITE_TAC [PAIR_EQ;EL_SET_EL]
  THEN INDEPENDENCE_TAC INDEP_A_UPDATE1 ;
    POP_ASSUM(\thm. DISJ_CASES_TAC thm)
  THEN EXPAND_REG_TAC
  THEN EXPAND_COND_TAC 15
  THEN ASM_REWRITE_TAC [PAIR_EQ;EL_SET_EL]
  THENL
    [ INDEPENDENCE_TAC INDEP_X_UPDATE1 ;
      INDEPENDENCE_TAC INDEP_Y_UPDATE1
    ]
  ]
  THEN ASM_REWRITE_TAC []
  THEN POP_ASSUM(\thm. ASM_REWRITE_TAC\[(REWRITE_RULE ([update_reg] @ asl) thm)] )))
);

%------ no overflow case ---------%

e( POP_ASSUM(\thm1. POP_ASSUM(\thm2.
  (ASSUME_TAC thm1 )
    THEN ASSUME_TAC( REWRITE_RULE [thm1] thm2) ))
  THEN NEXT_SYMB_EXEC_TAC 10 THEN DELETE_USTEP_TAC 9
  % from SHIFT_GOOD_DEST_TAC2  
  THEN ASSUM_LIST(\asl. DISJ_CASES_TAC (el 15 asl) )
  THENL [ EXPAND_REG_TAC
  THEN EXPAND_COND_TAC 15  % rewrite bitn term %
  THEN ASM_REWRITE_TAC [PAIR_EQ;EL_SET_EL]
  THEN INDEPENDENCE_TAC INDEP_A_UPDATE1 ;
    POP_ASSUM(\thm. DISJ_CASES_TAC thm)
THEN EXPAND_REG_TAC
THEN EXPAND_COND_TAC 15
THEN ASM_REWRITE_TAC [PAIR_EQ;EL_SET_EL]
THENL
[ INDEPENDENCE_TAC INDEP_X_UPDATE1 ;
  INDEPENDENCE_TAC INDEP_Y_UPDATE1
]
]
THEN ASM_REWRITE_TAC []
THEN POP_ASSUM(
  thm. (ASSUM_LIST(
    asl. REWRITE_TAC
    [(REWRITE_RULE ([update_reg] @ asl) thm)])
  )))
);;
Appendix H: PHASE LEVEL SPECIFICATION

File: def_phase.ml

Description: Defines the behavioral description of the phase level interpreter.

Modified by ETS to reflect block changes.

set_search_path (search_path() @ lib_dir_list);

loadf 'abstract';

system '/bin/rm phase_def.th';

new_theory 'phase_def';

map new_parent ['aux_def'; 'tuple'; 'regs_def'; 'ucode_def'; 'threeval'];

let rep_ty = abstract_type 'aux_def' 'opcode';

let phase_one_def = new_definition
('phase_one_def',
  "! (rep:'rep_ty) (regs:(*wordn)list) (mreg insreg din dout:*wordn)
    (ram:*memory) (b stop ovl:bool) (mar:*address) (res:*wordn)
    (mpc:btT) (mir:ucode) (urom:num->ucode) (rlatch mlatch:*wordn)
    (ph1 ph2 ph3:bool) (reset:bool).

  phase one rep (regs, mreg, insreg, din, dout, ram, b, stop, ovl, mar, res,
    mpc, mir, urom, rlatch, mlatch, ph1, ph2, ph3) (reset) =
    stop => (regs, mreg, insreg, din, dout, ram, b, T, ovl, mar, res,
      (F,F,F,F,F,F), mir, urom, rlatch, mlatch, F, F) |
      (regs, mreg, insreg, din, dout, ram, b, F, ovl, mar, res,
        mpc, urom (bt7_val mpc), urom, rlatch, mlatch, F, T, F) "
);

let phase_two_def = new_definition
('phase_two_def',
  "! (rep:'rep_ty) (regs:(*wordn)list) (mreg insreg din dout:*wordn)
    (ram:*memory) (b stop ovl:bool) (mar:*address) (res:*wordn)
    (mpc:btT) (mir:ucode) (urom:num->ucode) (rlatch mlatch:*wordn)"
phase_two rep (regs, mreg, insreg, din, dout, ram, b, stop, ovl, mar, res, mpc, mir, urom, rlatch, mlatch, phl, ph2, ph3) (reset) =
(regs, mreg, insreg, din,
(W mir => EL (bt2_val(Rfc mir => (Mrf mir) \ RSF rep insreg)\regs) | dout),
ram, b,
((FST(decode rep(opcode rep insreg, b))) \ (Dec_ctl mir)) \ /
(Seqctl mir = (F,F,T)) \ /
  (((FST(SND(decode rep(opcode rep insreg, b)))) = (F,F,T,T,F)) \ /
  (((FST(SND(decode rep(opcode rep insreg, b)))) = (F,F,T,T,T)) \ /
  ((MSF rep insreg) = (F,F)) ) \ /
(Seqctl mir = T,F,F) \ / ovl \ /
(Seqctl mir = T,F,T) \ / "valid_address rep res \ /
(Seqctl mir = T,T,F) \ /
( ((DSF rep insreg = (T,T,F)) \ (DSF rep insreg = (T,T,T))) \ /
  "(( DSF rep insreg = (T,F,F) \ "b) \ /
    ( (DSF rep insreg = (T,F,T)) \ b) ) \ /
    (DSF rep insreg = (F,T,T)) \ /
    (DSF rep insreg = (F,T,F)) \ /
    (DSF rep insreg = (F,F,F)) \ /
    (DSF rep insreg = (F,T,T)) ))) \ /
((R mir \ W mir) =>
  (Adrs mir => address rep insreg |
    address rep(EL p_reg regs)\mir),
res, mpc, mir, urom,
EL (bt2_val(Rfc mir => (Mrf mir) \ RSF rep insreg)\regs,
(Ms mir = F,F) => mreg |
  ((Ms mir = F,T) => wordn rep 1 \ pad rep(address rep insreg))).
F,F,
"((FST(decode rep(opcode rep insreg, b))) \ (Dec_ctl mir)) \ /
  (Seqctl mir = (F,F,T)) \ /
    (((FST(SND(decode rep(opcode rep insreg, b)))) = (F,F,T,T,F)) \ /
    (((FST(SND(decode rep(opcode rep insreg, b)))) = (F,F,T,T,T)) \ /
    ((MSF rep insreg) = (F,F)) ) \ /
  (Seqctl mir = T,F,F) \ / ovl \ /
  (Seqctl mir = T,F,T) \ / "valid_address rep res \ /

248
(Seqctl mir = T,T,F) /
( ((DSF rep insreg = (T,T,F)) \ (DSF rep insreg = (T,T,T))) /
  (~((DSF rep insreg = (T,F,F)) \ ~b) /
   (DSF rep insreg = (T,F,T)) / b)) /
  ((DSF rep insreg = (F,T,T)) /
   (DSF rep insreg = (T,F,F)) /
   (DSF rep insreg = (T,T,F)) /
   "valid_address rep res)) /

(Seqctl mir = T,T,T) /
( (DSF rep insreg = (F,T,T)) /
  (DSF rep insreg = (T,F,F)) /
  (DSF rep insreg = (T,F,T)) /
  (DSF rep insreg = (T,T,T)))")

%;

%---------------------------------------------------------------
has let definitions. takes a long time to load, so replaced
it by HOL-expanded definition.
%---------------------------------------------------------------

let rselect = bt2_val((Rfc mir) => (Mrf mir) | RSF rep insreg) in
let r_out = (EL rselect regs) in
let new_dout = (W mir) => r_out | dour) in
let bad_res = "(valid_address rep res) in
let df = (DSF rep insreg) in
  let pdest = ((df=(F,T,T)) /
    (df=(T,F,F)) /
    (df=(T,F,T)) /
    (df=(T,T,F)) /
    (df=(T,T,T))) in
  let skip = ((df=(T,F,F)) \ ~b) /
    ((df=(T,F,T)) \ b) in
  let bad_rdest = ((df=(T,T,F)) /
    (df=(T,T,T)) /
    (df=(T,F,T)) /
    (df=(T,F,F)) /
    (df=(T,T,T))) in
  let bad_dest = ((df=(F,T,T)) /
    (df=(T,F,F)) /
    (df=(T,F,T)) /
    (df=(T,T,F)) /
    (df=(T,T,T))) in
  let seq_case4 = ((Seqctl mir) = (T,F,F)) in
  let seq_case5 = ((Seqctl mir) = (T,F,T)) in
  let seq_case6 = ((Seqctl mir) = (T,T,F)) in
  let seq_case7 = ((Seqctl mir) = (T,T,T)) in
  let msl_stop = ((seq_case4 \ ovl)
    (seq_case5 \ bad_res)
    (seq_case6 \ (bad_rdest \ (~skip / pdest \ bad_res))
    (seq_case7 \ bad_dest) in
  let new_stop = (((FST (decode rep (opcode rep insreg, b)))
    \ (Dec_ctl mir))
  \ msl_stop) in
let adr_out = ((Adrs mir) => (address rep insreg) |
(address rep (EL p_reg regs))) in
let new_mar = (((R mir) \ (W mir)) => adr_out | mar) in
let new_rLatch = r_out in
let new_mlatch = (((Ms mir) = (F,F)) => mreg |
((Ms mir) = (F,T)) => (wordn rep 1) |
(pad rep (address rep insreg))) in
(regs, mreg, insreg, din, new_dout, ram, 'b, new_stop, ovl, new_mar,
res, mpc, mir, urom, new_rLatch, new_mlatch, F, F, "new_stop")
-----------------------------------------------------------------------%

let phase_three_def = new_definition
('phase__three_def',
"'(rep:'rep_ty) (regs:(*wordn)list) (mreg insreg din dout:*wordn)
(ram:*memory) (b stop ovl:bool) (mar:*address) (res:*wordn)
(mpc:bt7) (mir:ucode) (urom:num->ucode) (rlatch mlatch:*wordn)
phase_three rep(regs, mreg, insreg, din, dout, ram, b, stop, ovl, mar, res,
mpc, mir, urom, rlatch, mlatch, phi, ph2, ph3) (reset) =

((Re mir =>
((Dfc mir \ (Mdf mir = (T,T,F)) \ (Mdf mir = (T,T,T))) =>
regs | update_reg regs
(Dfc mir => (Mdf mir) | DSF rep insreg) b
(((Aluctl mir = F,F,F,F) \ (Aluctl mir = F,F,T,F)) =>
mlatch |
((Aluctl mir = F,F,F,T) =>
rlatch |
((Aluctl mir = F,F,T,T) =>
neg rep mlatch |
(((Aluctl mir = F,T,F,F) \ (Aluctl mir = F,T,F,T)) =>
add rep(rlatch,mlatch) |
(((Aluctl mir = F,T,F,F) \ (Aluctl mir = F,T,F,T)) =>
sub rep(rlatch,mlatch) |
((Aluctl mir = T,F,F,F) =>
bxor rep(rlatch,mlatch) |
((Aluctl mir = T,F,F,T) =>
band rep(rlatch,mlatch) |
((Aluctl mir = T,F,T,F) =>
bnor rep(rlatch,mlatch) |
((Aluctl mir = T,F,T,T) =>
band rep(rlatch,bnot rep mlatch) |
((Aluctl mir = T,T,F,F) =>
shr rep rlatch |

250
\((\text{Aluctl mir} = T,T,F,T) \Rightarrow\)
\(\text{shrb rep}(\text{rlatch},b)\ |
\((\text{Aluctl mir} = T,T,T,F) \Rightarrow\)
\(\text{shl rep rlatch |}
\(\text{shlb rep}(\text{rlatch},b))))))))\)
\(\text{regs),}
\(\text{(De mir} \Rightarrow\)
\(\text{(Ds mir} \Rightarrow \text{mreg | din) |}
\(\text{(Re mir} \land \text{Dfc mir} \land\)
\(\text{((bt3_val}(\text{Dfc mir} \Rightarrow(\text{Mdf mir}) \land \text{DSF rep insreg})=6)) \Rightarrow\)
\(\text{((Aluctl mir} = F,F,F,F) \land (\text{Aluctl mir} = F,F,T,F) \Rightarrow\)
\(\text{mlatch |}
\(\text{((Aluctl mir} = F,F,F,T) \Rightarrow\)
\(\text{rlatch |}
\(\text{((Aluctl mir} = F,F,T,T) \Rightarrow\)
\(\text{neg rep mlatc |}
\(\text{((Aluctl mir} = F,T,F,F) \land (\text{Aluctl mir} = F,T,F,T) \Rightarrow\)
\(\text{add rep}(\text{rlatch},\text{mlatch}) |\)
\(\text{((Aluctl mir} = F,T,F,F) \land (\text{Aluctl mir} = F,T,F,T) \Rightarrow\)
\(\text{sub rep}(\text{rlatch},\text{mlatch}) |\)
\(\text{((Aluctl mir} = T,F,F,F) \Rightarrow\)
\(\text{bxor rep}(\text{rlatch},\text{mlatch}) |\)
\(\text{((Aluctl mir} = T,F,F,T) \Rightarrow\)
\(\text{band rep}(\text{rlatch},\text{mlatch}) |\)
\(\text{((Aluctl mir} = T,F,T,F) \Rightarrow\)
\(\text{bnor rep}(\text{rlatch},\text{mlatch}) |\)
\(\text{((Aluctl mir} = T,F,T,T) \Rightarrow\)
\(\text{band rep}(\text{rlatch},\text{bnot rep mlatch}) |\)
\(\text{((Aluctl mir} = T,T,F,F) \Rightarrow\)
\(\text{shr rep rlatch |}
\(\text{((Aluctl mir} = T,T,F,T) \Rightarrow\)
\(\text{shrb rep}(\text{rlatch},b) |)
\(\text{((Aluctl mir} = T,T,T,F) \Rightarrow\)
\(\text{shl rep rlatch |}
\(\text{shlb rep}(\text{rlatch},b))]))))))\)
\(\text{mreg})\),
\(\text{(De mir} \Rightarrow\)
\(\text{(Ds mir} \Rightarrow \text{din | insreg) |}
\(\text{(Re mir} \land \text{Dfc mir} \land\)
\(\text{((bt3_val}(\text{Dfc mir} \Rightarrow(\text{Mdf mir}) \land \text{DSF rep insreg})=7)) \Rightarrow\)
\(\text{join rep (opcode rep insreg, address rep}
\(\text{((Aluctl mir} = F,F,F,F) \land (\text{Aluctl mir} = F,F,T,F) \Rightarrow\)
\(\text{mlatch |}
\(\text{((Aluctl mir} = F,F,F,T) \Rightarrow\)

251
rlatch |
((Aluctl mir = F,F,T,T) =>
 neg rep m latch |
(((Aluctl mir = F,T,F,F) \ (Aluctl mir = F,T,F,T)) =>
 add rep(r latch, m latch) |
(((Aluctl mir = F,T,T,F) \ (Aluctl mir = F,T,T,T)) =>
 sub rep(r latch, mlatch) |
((Aluctl mir = T,F,F,F) =>
 bxor rep(r latch, mlatch) |
((Aluctl mir = T,F,F,T) =>
 band rep(r latch, mlatch) |
((Aluctl mir = T,F,T,F) =>
 bnor rep(r latch, mlatch) |
((Aluctl mir = T,F,T,T) =>
 band rep(r latch, bnot rep m latch) |
((Aluctl mir = T,T,F,F) =>
 shr rep rlatch |
((Aluctl mir = T,T,F,T) =>
 shrb rep(r latch, b) |
((Aluctl mir = T,T,T,F) =>
 shl rep rlatch |
((Aluctl mir = T,T,T,T) =>
 shlb rep(r latch, b)))))))) |)
insreg),
(R mir => (Io mir => fetchio rep(ram,mar) | fetch rep(ram,mar) | dim),
dout,
(W mir=>(Io mir=>storeio rep(ram,mar,dout) | store rep(ram,mar,dout)) | ram),
((Aluctl mir = F,F,T,F) =>
 bcmp rep(r latch, m latch,b,FSF rep insreg) |
((Aluctl mir = F,T,F,F) =>
 addp rep(r latch, m latch,add rep(r latch, m latch)) |
((Aluctl mir = F,T,F,T) =>
 subp rep(r latch, m latch,sub rep(r latch, m latch)) |
((Aluctl mir = T,T,F,F) =>
 bitO rep rlatch |
((Aluctl mir = T,T,T,T) => bitn rep rlatch | b)))))),
F,
(((Aluctl mir = F,T,F,F) \ (Aluctl mir = F,T,F,T)) =>
 aovfl rep(r latch, m latch,add rep(r latch, m latch)) |
(((Aluctl mir = F,T,T,F) \ (Aluctl mir = F,T,T,T)) =>
 sovfl rep (r latch, m latch,sub rep(r latch, m latch)) |
((Aluctl mir = T,T,T,F) => bitn rep rlatch | F))),
mar,
(((Aluctl mir = F,F,F,F) \ (Aluctl mir = F,F,T,F)) =>
m latch |
((Aluctl mir = F,F,F,T) =>
  r latch |
((Aluctl mir = F,F,T,T) =>
  neg rep mlatch |
  (((Aluctl mir = F,T,F,F) \ (Aluctl mir = F,T,F,T)) =>
  add rep(r latch,mlatch) |
  (((Aluctl mir = F,T,F,F) \ (Aluctl mir = F,T,F,T)) =>
  sub rep(r latch,mlatch) |
  ((Aluctl mir = T,F,F,F) =>
  bxor rep(r latch,mlatch) |
  ((Aluctl mir = T,F,F,T) =>
  band rep(r latch,mlatch) |
  ((Aluctl mir = T,F,T,F) =>
  bnor rep(r latch,mlatch) |
  ((Aluctl mir = T,F,T,T) =>
  add rep(r latch,mlatch) |
  ((Aluctl mir = T,T,F,F) =>
  shr rep rlatch |
  ((Aluctl mir = T,T,F,T) =>
  shr rep rlatch |
  ( (Aluctl mir = T,T,T,F) =>
  band rep rlatch |
  ( (Aluctl mir = T,T,T,T) =>
  shlb rep rlatch |
  ((Aluctl mir = T,T,F,F) =>
  shr rep rlatch |
  ( (Aluctl mir = T,T,F,T) =>
  shr rep rlatch |
  ( (Aluctl mir = T,T,T,F) =>
  shr rep rlatch |
  ( (Aluctl mir = T,T,T,T) =>
  shlb rep rlatch ))))))))))
((Seqctl mir = F,F,T) \ SND(SND(decode rep(opcode rep insreg,b))) \ /
( Seqctl mir = F,T,F) \ /
( Seqctl mir = F,T,T) =>
  (((Seqctl mir = F,F,T)
  /\ (FST(SND(dec ord rep(opcode rep insreg,b)))) = (F,F,T,T,F) | (FST(SND(dec ord rep(opcode rep insreg,b)))) = (F,F,T,T,T))
  /\ (MSF rep insreg = (F,F))
  /\ SND(SND(de code rep(opcode rep insreg,b)))
 ) =>
bt7_ival((bt7_val(M addr mir)) + (bt2_val(MSF rep insreg))) |
  ((Seqctl mir = F,T,F) =>
bt7_ival
  ((bt7_val(Maddr mir)) +
  (bt5_val(FST(SND(de code rep(opcode rep insreg,b)))))) |
bt7_ival((bt7_val mpc) + 1)),
mir, urom, r latch, mlatch, T,F,F"
let alu_case0 = ((Aluctl mir) = (F,F,F,F)) in
  let alu_case1 = ((Aluctl mir) = (F,F,F,T)) in
  let alu_case2 = ((Aluctl mir) = (F,F,T,F)) in
  let alu_case3 = ((Aluctl mir) = (F,F,T,T)) in
  let alu_case4 = ((Aluctl mir) = (F,T,F,F)) in
  let alu_case5 = ((Aluctl mir) = (F,T,F,T)) in
  let alu_case6 = ((Aluctl mir) = (F,T,T,F)) in
  let alu_case7 = ((Aluctl mir) = (F,T,T,T)) in
  let alu_case8 = ((Aluctl mir) = (T,F,F,F)) in
  let alu_case9 = ((Aluctl mir) = (T,F,F,T)) in
  let alu_case10 = ((Aluctl mir) = (T,F,T,F)) in
  let alu_case11 = ((Aluctl mir) = (T,F,T,T)) in
  let alu_case12 = ((Aluctl mir) = (T,T,F,F)) in
  let alu_case13 = ((Aluctl mir) = (T,T,F,T)) in
  let alu_case14 = ((Aluctl mir) = (T,T,T,F)) in
  let alu_case15 = ((Aluctl mir) = (T,T,T,T)) in
  let sum = (add rep (rlatch, mlatch)) in
  let diff = (sub rep (rlatch, mlatch)) in
  let result = ((alu_case0) \ (alu_case2)) \ m latch | alu_case1 => rlatch | alu_case3 => (neg rep mlatch) | (alu_case4 \ alu_case5) => sum | (alu_case6 \ alu_case7) => diff | alu_case8 => (bxor rep (rlatch, mlatch)) | alu_case9 => (band rep (rlatch, mlatch)) | alu_case10 => (bnor rep (rlatch, mlatch)) | alu_case11 => (band rep (rlatch, bnot rep mlatch)) | alu_case12 => (shr rep rlatch) | alu_case13 => (shrb rep (rlatch, b)) | alu_case14 => (shl rep rlatch) |
  (shlb rep (rlatch, b)) in

let w_reg = ((Dfc mir) => (Mdf mir) | DSF rep insreg) in
let new_regs =
  ((Re mir) => ((Dfc mir) \ (Mdf mir = (T,T,F)) \ (Mdf mir = (T,T,T))) =>.reg s | update_reg regs w_reg b result) |
  regs) in
let new_mreg =
  ( (De mir) => ((Ds mir) => mreg | din) | ((Re mir) \ (Dfc mir) \ (bt3_val(w_reg)=6))) =>
result | mreg) in

let new_insreg =
    ( (De mir) => ((Ds mir) => din | insreg) |
        (((Re mir) \ (Dc mir) \ (bt3_val(w_reg)=7)) =>
            (join rep (opcode rep insreg, address rep result)) |
        insreg) ) in

let new_din = ((R mir) => ((Io mir) => fetchio rep (ram, mar) |
    fetch rep (ram, mar)) |
    din) in

let new_ram = ((W mir) => ((Io mir) => storeio rep (ram, mar, dout) |
    store rep (ram, mar, dout)) |
    ram) in

let new_b = (alu_case2 =>
    (bcmp rep (rlatch, mlatch, b, FSF rep insreg)) |
    alu_case4 => (addp rep (rlatch, mlatch, sum)) |
    alu_case6 => (subp rep (rlatch, mlatch, diff)) |
    alu_case13 => (bitO rep rlatch) |
    alu_case15 => (bitn rep rlatch) |
    b ) in

let new_ovl = ((alu_case4 \ alu_case5) =>
    (aovfl rep (rlatch, mlatch, sum)) |
    (alu_case6 \ alu_case7) =>
        (sovfl rep (rlatch, mlatch, diff)) |
    alu_case14 => (bitn rep rlatch) |
    F ) in

let new_res = result in

let seq_case1 = ((Seqctl mir) = (F,F,T)) in
    let seq_case2 = ((Seqctl mir) = (F,T,F)) in
    let seq_case3 = ((Seqctl mir) = (F,T,T)) in

let reqm = (SND(SND(decode rep (opcode rep insreg, b)))) in
let opc = (FST(SND(decode rep (opcode rep insreg, b)))) in
let jaddr = ((seq_case1 \ reqm) =>
    (bt7_ival ((bt7_val (Maddr mir))+(bt2_val(MSF rep insreg)))) |
    seq_case2 =>
        (bt7_ival ((bt7_val (Maddr mir))+(bt5_val opc))) |
    seq_case3 => (Maddr mir) |
        (F,F,F,F,F,F,F) ) in

let muxmc = ((seq_case1 \ reqm) \ seq_case2 \ seq_case3) in

let new_mpc = (muxmc => jaddr | bt7_ival (bt7_val mpc + 1)) in

(new_regs, new_mreg, new_insreg, new_din, dout, new_ram, new_b, F,
new_ovl, mar, new_res, new_mpc, mir, urom, rlatch, mlatch, T, F, F)"

-------------------------------------------------------------%
Selector function on phase level state for the phase level counter.

\[ \text{let GetPhaseClock = new_definition}
('GetPhaseClock',
"! (regs:(*wordn)list) (mreg insreg din dout:*wordn)
 (ram:*memory) (b stop ovl:bool) (mar:*address) (res:*wordn)
 (mpc:bt7) (mir:ucode) (urom:num->ucode) (rlatch mlatch:*wordn)
 GetPhaseClock (regs, mreg, insreg, din, dout, ram, b, stop, ovl, mar,
 res, mpc, mir, urom, rlatch, mlatch, phl, ph2, ph3) (reset) =
 (ph2 => TWO |
  ph3 => THREE |
  ONE)");
\]

\[ \text{Gives the number of EBM cycles to implement one phase level cycle.}
\]

\[ \text{let PhaseLevelCycles = new_definition}
('PhaseLevelCycles',
"! t:triple.
 PhaseLevelCycles t = 1"
);;
\]

\[ \text{let PhaseClockBegin = new_definition}
('PhaseClockBegin',
"PhaseClockBegin = ONE"
);;
\]

\[ \text{let Phase_Substate = new_definition}
('Phase_Substate',
"! (regs:*wordn)list) (mreg insreg din dout:*wordn)
 (ram:*memory) (b stop ovl:bool) (mar:*address) (res:*wordn)
 (mpc:bt7) (mir:ucode) (urom:num->ucode) (rlatch mlatch:*wordn)
 (ph1 ph2 ph3:bool) (reset:bool).
 Phase_Substate (regs, mreg, insreg, din, dout, ram, b, stop, ovl, mar,
 res, mpc, mir, urom, rlatch, mlatch, phi, ph2, ph3) =
\]

256
(regs,mreg,insreg,din,dout,ram,b,stop,ovl,mar,res,mpc)

"');

%------------------------------------------------------
'I' serves as the substate function since the state
of the phase level is equivalent to the phase of the EBM.

'I' also serves as the subenv function since the set of external
lines in the phase level is the same as the set of external
lines in the EBM.
------------------------------------------------------%

close_theory();


File: mk_phase.ml

Description: Defines the phase level interpreter in terms of the definitions in block_def.th, phase_def.th, and gen_I.th.

Proves the lemmas meeting the theory obligations for the abstract theory gen_I.th and instantiates a proof of the phase level in terms of the EBM.

---

set_search_path (search_path() @ lib_dir_list);

loadf 'abstract';

system '/bin/rm phase.th';

new_theory 'phase';

map new_parent ['gen_I';'phase_def';'block_def'];

load_definitions 'threeval';

load_theorems 'threeval';

let time_shift = definition 'gen_I' 'time_shift';

let GetPhaseClock = definition 'phase_def' 'GetPhaseClock';

let PhaseLevelCycles = definition 'phase_def' 'PhaseLevelCycles';

let phase_one_def = definition 'phase_def' 'phase_one_def';

let phase_two_def = definition 'phase_def' 'phase_two_def';

let phase_three_def = definition 'phase_def' 'phase_three_def';

let GetEBMClock = definition 'block_def' 'GetEBMClock';

let ESM_Start = definition 'block_def' 'EBM_Start';

let EBM_expanded = theorem 'block_def' 'EBM_expanded';

loadf 'tuple';

let rep_ty = abstract_type 'aux_def' 'opcode';

let I_rep_ty = abstract_type 'gen_I' 'Impl';

let Phase_state =
  "%(wordn)list # wordn # wordn # wordn # wordn # memory # bool #
  bool # bool # address # wordn # bt7 # ucode # (num -> ucode) #"
let Phase_env = ":bool";;
let EBM_state = Phase_state;;
let EBM_env = Phase_env;;

%-----------------------------------------------------------------------%
% We might as well do this now, we'll have to do it sooner or later.
%-----------------------------------------------------------------------%

%-----------------------------------------------------------------------%
let phase_two_expanded =
  EXPAND_LET_RULE phase_two_def;;

let phase_three_expanded =
  EXPAND_LET_RULE phase_three_def;;

%-----------------------------------------------------------------------%

%-----------------------------------------------------------------------%
Define the phase level interpreter in terms of the generic
interpreter definition.
%-----------------------------------------------------------------------%

let Phase_I_def = new_definition
('Phase_I_def',
"! (rep:'rep_ty) (s:time->'Phase_state) (e:time->'Phase_env).
  Phase_I rep s e =
    INTERP
      ([ONE,phase_one rep;
         TWO,phase_two rep;
         THREE,phase_three rep],
       triple_value,
       (GetPhaseClock:'Phase_state->'Phase_env->triple),
       (PhaseLevelCycles:triple->num),
       (I:'EBM_state->'Phase_state),
       (I:'EBM_env->'Phase_env), EBM rep,
       (GetEBMClock:'EBM_state->'EBM_env->bool),
       EBM_Start, @x:one.F) s e"
);;

let PHASE_I = save_thm
('PHASE_I',
BETA_RULE (EXPAND_LET_RULE
  (instantiate_abstract_definition 'gen_I' 'INTERP' Phase_I_def)));;

let Phase_I_IMPL_IMP_DEF = new_definition
('Phase_I_IMPL_IMP_DEF',
"! (rep: 'rep_ty) s' e'.
Phase_I_IMPL_IMP rep s' e' =

IMPL_IMP
([ONE,phase_one rep;
  TWO,phase_two rep;
  THREE,phase_three rep],
triple.value,
(GetPhaseClock: 'Phase_state -> 'Phase_env -> triple),
(PhaseLevelCycles: triple->num),
(I: 'EBM_state->'Phase_state),
(I: 'EBM_env->'Phase_env), EBM rep,
(GetEBMClock: 'EBM_state->'EBM_env->bool),
EBM_Start, \x:one. F) s' e''
);

let Phase_I_IMPL_IMP =
  let Phase_I_EXT =
    CONV_RULE (TOP_DEPTH_CONV FUN_EQ_CONV) Phase_I_IMPL_IMP_DEF in
    (REWRITE_RULE [I_THM]
    (BETA_RULE (EXPAND_LET_RULE
      (instantiate_abstract_definition
        'gen_I' 'IMPL_IMP' Phase_I_EXT))));

%---------------------------

We need to establish the first theory obligation for the abstract
theory for a generic interpreter. First, we
will prove Phase_I_IMPL_IMP applies to each of the phases and
then use these results to establish that Phase_I_IMPL_IMP applies
to EVERY instruction (i.e. the first theory obligation.

%---------------------------

A lemma needed for rewriting

%---------------------------

let cond3_def = new_definition
  ('cond3_def',

  "!c1 c2 . cond3_def c1 c2 =
    (c1 => TWO |
     c2 => THREE |
     ONE)"
  );

let xx = prove_constructors_distinct triple;;

let cond3_lemma = prove_thm
  ('cond3_lemma',

  "! c1 c2 . ((cond3_def c1 c2 = TWO) ==> c1) /

  260
let COND_NULL_LEMMA = TAC_PROOF
  (([], "! b (c: *).
   (b => c(c) = c"),
   REPEAT GEN_TAC
   THEN BOOL_CASES_TAC "b"
   THEN REWRITE_TAC[])
);

%---------------------------------------------------------------------
PHASE_EBM_TAC is used to prove that the individual phases
satisfy Phase_I_IMPL_IMP.

%---------------------------------------------------------------------
let PHASE_EBM_TAC =
  PURE_ONCE_REWRITE_TAC [Phase_I_IMPL_IMP]
  THEN REPEAT GEN_TAC
  THEN BETATAC
  THEN REWRITE_TAC [GetPhaseClock; PhaseLevelCycles;
    GetEBMClock; EBM_Start; phase_one_def;
    phase_two_def; phase_three_def]
  THEN SUBST_TAC [EBM_expanded]
  THEN REPEAT STRIP_TAC
  THEN POP_ASSUM_LIST (asl. (MAP_EVERY (STRIP_ASSUME_TAC o SPEC_ALL) asl))
  THEN POP_ASSUM_LIST (asl. (MAP_EVERY (STRIP_ASSUME_TAC o SPEC_ALL) asl));

let PHASE_ONE_EBM_LEMMA = TAC_PROOF
  (([], "!(rep:*rep_ty) (regs:time->*wordn)list
   (mreg insreg din dout:time->*wordn) (ram:time->*memory)
   (b stop ovl:time->bool) (mar:time->*address) (res:time->*wordn)
   (mpc:time->bit7) (mir:time->ucode) (urom:num->ucode)
   (rlatch mlatch:time->*wordn) (ph1 ph2 ph3:time->bool)
   (reset:time->bool).
   Phase_I_IMPL_IMP rep
   (\t:num.
    (regs t, mreg t, insreg t, din t, dout t, ram t, b t, stop t,
     ovl t, mar t, res t, mpc t, mir t, urom, rlatch t, mlatch t,
let PHASE_TWO_EBN_LEMMA = TAC_PROOF
(([]),
"!(rep:'rep_ty) (regs:time->(wordn)list)
  (mreg insreg din dout:time->wordn) (ram:time->memory)
  (b stop ovl:time->bool) (mar:time->address) (res:time->wordn)
  (mpc:time->bt7) (mir:time->ucode) (urom:num->ucode)
  (rlatch mlatch:time->wordn) (ph1 ph2 ph3:time->bool)
  (reset:time->bool).
Phase_I_IMPL_IMP
((
  t. (regs t, mreg t, insreg t, din t, dout t, ram t, b t, stop t,
    ovl t, mar t, res t, mpc t, mir t, urom, rlatch t, mlatch t,
    ph1 t, ph2 t, ph3 t))

  (t. (reset t))
)(TWO,phase_two rep)"),
PHASE_EBM_TAC
REWRITE_TAC[PAIR_EQ] THEN

...
let PHASE_THREE_EBM_LEMMA = TAC_PROOF
(([]),
 "!(rep:'rep_ty) (regs:time->(*wordn)list)
 (mreg insreg din dout:time->*wordn) (ram:time->memory)
 (b stop ovl:time->bool) (mar:time->*address) (res:time->*wordn)
 (mpc:time->bt7) (mir:time->ucode) (urom:num->ucode)
 (rlatch mlatch:time->*wordn) (ph1 ph2 ph3:time->bool)
 (reset:time->bool).
 Phase_I_IMPL_IMP re

 Ph

 PHASE_EBM_TAC THEN
 REWRITE_TAC[PAIR_EQ] THEN
 POP_ASSUM \thm. STRIP_ASSUME_TAC (MULTI_MP
 (CONJUNCTS (SPECL ["(ph2 t):bool"; "(ph3 t):bool"]
 (REWRITE_RULE [cond3_def] cond3_lemma))) thm)) THEN
 ASSUM_LIST \asl. STRIP_ASSUME_TAC
 (REWRITE_RULE [el 1 asl] (el 8 asl))) THEN
 POP_ASSUM_LIST \asl. (MAP_EVERY
 (CHECK_ASSUME_TAC o (REWRITE_RULE
 [(el 1 asl); (el 2 asl); (el 3 asl); (el 4 asl)]) asl)) THEN
 ASM_REWRITE_TAC[] THEN
 REWRITE_TAC[COND_NULL_LEMMA]
 );;

 let PHASE_THREE_EBM_LEMMA = TAC_PROOF
 (([]),
  
  PHASE_EBM_TAC THEN
  REWRITE_TAC[PAIR_EQ] THEN
  POP_ASSUM \thm. STRIP_ASSUME_TAC (MULTI_MP
  (CONJUNCTS (SPECL ["(ph2 t):bool"; "(ph3 t):bool"]
   (REWRITE_RULE [cond3_def] cond3_lemma))) thm)) THEN
  ASSUM_LIST \asl. STRIP_ASSUME_TAC
   (REWRITE_RULE [el 1 asl] (el 8 asl))) THEN
  POP_ASSUM_LIST \asl. (MAP_EVERY
   (CHECK_ASSUME_TAC o (REWRITE_RULE
    [(el 1 asl); (el 2 asl); (el 3 asl); (el 4 asl)]) asl)) THEN
   ASSUM_LIST \asl. REWRITE_TAC[SYM
   (REWRITE_RULE [el 43 asl] (el 8 asl))) THEN
   POP_ASSUM_LIST \asl. MAP_EVERY (thm.

   let rat = (((fst o dest_var o rator o fst o dest_eq)
    (concl thm) ? 'xxx') and
   ran = (((fst o dest_var o rand o fst o dest_eq)
    (concl thm) ? 'xxx') in
   if ((mem rat (words 'result')) & (mem ran (words 't')))
    then ALL_TAC else ASSUME_TAC thm) asl) THEN
The first obligation of the abstract interpreter theory

\begin{verbatim}
let Phase_I_EVERY_IMPL_IMP = TAC_PROOF
(([],
  "!(rep:'rep_ty) (regs:time->(wordn)list)
  (mreg insreg din dout:time->wordn) (ram:time->memory)
  (b stop ovl:time->bool) (mar:time->address) (res:time->wordn)
  (mpc:time->b7) (mir:time->ucode) (urom:num->ucode)
  (rlatch mlatch:time->wordn) (ph1 ph2 ph3:time->bool)
  (reset:time->bool).
  EVERY (Phase_I_IMPL_IMP rep
  (\t. (regs t, mreg t, insreg t, din t, dout t, ram t, b t, stop t,
  ovl t, mar t, res t, mpc t, mir t, urom, rlatch t, mlatch t,
  ph1 t, ph2 t, ph3 t))
  (\t. (reset t)))
  [ONE,phase_one rep;
  TWO,phase_two rep;
  THREE,phase_three rep])",)
  REWRITE_TAC [EVERY_DEF]
  THEN REPEAT STRIP_TAC
  THEN FIRST [
    MATCH_ACCEPT_TAC PHASE_ONE_EBM_LF;
    MATCH_ACCEPT_TAC PHASE_TWO_EBM_LEM/_A;
    MATCH ACCEPT_TAC PHASE_THREE_EBM_LEMMA
  ]
);;

let Phase_I_EVERY_LEMMA = (SPEC_ALL
  (PURE_ONCE_REWRITE_RULE [Phase_I_IMPL_IMP_DEF] Phase_I_EVERY_IMPL_IMP));;
\end{verbatim}

The second obligation of the abstract interpreter theory

\begin{verbatim}
let Phase_I_LENGTH_LEMMA = TAC_PROOF
(([],
  "! k:triple. triple_value k < (LENGTH [ONE,phase_one (rep:'rep_ty);
  TWO,phase_two rep;
  THREE,phase_three rep])")
  MATCH_ACCEPT_TAC triple_LENGTH_LEMMA
);;
\end{verbatim}
The third obligation of the abstract interpreter theory

let triple_cases = prove_cases_thm (prove_induction_thm triple);

let Phase_I_KEY_LEMA = TAC_PROOF
  (([],
    "!k:triple . k = (FST (EL (triple_value k)[ONE,phase_one (rep:'rep_ty);
    TWO,phase_two rep;
    THREE,phase_three rep]))"),
  REPEAT GEN_TAC
  THEN STRUCT_CASES_TAC (SPEC "k:triple" triple_cases)
  THEN REWRITE_TAC (CONJUNCTS triple_VALUE_LEMA)
  THEN CONV_TAC (TOP_DEPTH_CONV num_CONV)
  THEN KEWITE_TAC [EL;FST;HD;TL]);;

Get the instantiation

let theorem_list =
  instantiate_abstract_theorems
  'gen_I'
  [Phase_I_EVERY_LEMA;
   Phase_I_LENGTH_LEMA;
   Phase_I_KEY_LEMA]
  ['rep:rep_ty',
   "ONE,phase_one (rep:'rep_ty);
   TWO,phase_two rep;
   THREE,phase_three rep],
   triple_value, (GetPhaseClock:'Phase_state->Phase_env->triple),
   PhaseLevelCycles, (I:'EBM_state->Phase_state),
   (I:'EBM_env->Phase_env),
   EBM rep, (GetEBMClock:'EBM_state->EBM_env->bool), EBM_Start);]
  ['t:time',
   "reset t):time->EBM_env);
   's':time->EBM_env"
  [
   (reg t, mreg t, instr t, din t, dout t, ram t,
   b t, stop t, ovl t, mar t, res t, mpc t, mir t, urom,
   r latch t, mlatch t, phl t,
   ph2 t, ph3 t):time->EBM_state);];
Timeshift doesn't mean anything at this level since they share a clock.

let TIME_SHIFT_DEGENERATE_LEMMA = TAC_PROOF

let correct_lemma = snd(hd theorem_list);

let EBM_IMPL_PHASE_LEMMA = save_thm

close_theory();
Appendix I: ELECTRONIC BLOCK LEVEL

%---------------------------------------------------------------%
File: defRegs.ml
Description: Register file definitions
%---------------------------------------------------------------%
set_search_path (search_path() @ lib_dir_list);

system '/bin/rm regs_def.th';
new_theory 'regs_def';
map new_parent ['aux_def'; 'aux_thms'];
%---------------------------------------------------------------%
Define selectors for register file
%---------------------------------------------------------------%
let A = new_definition ('A', "a_reg = 0");
let X = new_definition ('X', "x_reg = 1");
let Y = new_definition ('Y', "y_reg = 2");
let P = new_definition ('P', "p_reg = 3");
%---------------------------------------------------------------%
Define mutators for the register file
%---------------------------------------------------------------%
let update_reg = new_definition ('update_reg',
"! (registers:*wordnlist) (n:bt3) b value.
update_reg registers n b value =
(((n=(F,F,F)) \ (n=(F,F,T)) \ (n=(F,T,F))) =>
  SET_EL (bt3_val n) registers value |
  (n=(F,T,T)) =>
  SET_EL p_reg registers value |
  (((n=(T,F,F)) \ b) \ ((n=(T,F,T)) \ ~b)) =>
  SET_EL p_reg registers value |
  registers))");
close_theory();
%-----------------------------------------------

File: def_block.ml

Description: Defines the behavioral description of the electronic block model.

Modified by ETS:

The sequence control logic now recognizes the stop case where the pc, io space or is the target but, it is not valid.

7/17 the register block now also receives the b flag value which must be passed to update_reg. The datapath was changed accordingly.

the msl also receives b and control unit, ebm
9/7 MSL stops for writeio or write with mf = (F,F)

-----------------------------------------------%

set_search_path (search_path() @ lib_dir_list);;
loadf 'abstract';;
system '/bin/rm block_def.th';;
new_theory 'block_def';;
map new_parent ['regs_def'; 'ucode_def'; 'tuple'];;
let rep_ty = abstract_type 'aux_def' 'opcode';;

%------------------------------------------------

Ground
%-------------------------------------------------%

let GND = new_definition
('GND',
"! out . GND out = (out = F)"
);

%------------------------------------------------

Mux which selects one of source register selects from instn and microinstn
%-------------------------------------------------%

let MUXR_SPEC = new_definition
('MUXR_SPEC',
"! ctl (a:bt2) b c .
MUXR_SPEC ctl a b c =
c = (ctl => a | b)"
)

268
Mux which selects one of destination register selects from instn and microinstn

```
let MUXD_SPEC = new_definition
('MUXD_SPEC',
"! ctl (a:bt3) b c .
MUXD_SPEC ctl a b c =
c = (ctl => a | b)"
);
```

Mux which selects addresses of next microinstruction

```
let MUXM_SPEC = new_definition
('MUXM_SPEC',
"! ctl (a:bt7) b c .
MUXM_SPEC ctl a b c =
c = (ctl => a | b)"
);
```

Register specification - *wordn (MLATCH, RLATCH, RES)

```
let REG_SPEC = new_definition
('REG_SPEC',
"! (i:time->*wordn) Id out .
REG_SPEC i Id out =
(! t:time . out(t+1) = Id t => i t
| out t)"
);
```

Flipflop (1-bit register) (B)

```
let FF_SPEC = new_definition
('FF_SPEC',
"! (in:time->bool) (Id:time->bool) (q:time->bool) .
FF_SPEC in Id q =
! t:num . q(t+1) = ((Id t) => in t | q t)"
);
```

Register with enable input - *wordn (DIN, DOUT)
let REG_EN_SPEC = new_definition
 ('REG_EN_SPEC',
   "! set clk (in:time->*wordn) out .
   REG_EN_SPEC set clk in out =
   !:time. out (t+1) = ((set t) \ (clk t)) => in t | out t"
 );;

%---------------------------------------------------------------------------
Register with enable input - *address (MAR)
%---------------------------------------------------------------------------

let MAR_SPEC = new_definition
 ('MAR_SPEC',
   "! set clk (in:time-*address) out .
   MAR_SPEC set clk in out =
   !:time. out (t+1) = ((set t) \ (clk t)) => in t | out t"
 );;

%---------------------------------------------------------------------------
PHASE CLOCK
%---------------------------------------------------------------------------

let PHASE_CLOCK_SPEC = new_definition
 ('PHASE_CLOCK_SPEC',
   "! dis p1 p2 p3.
   PHASE_CLOCK_SPEC dis p1 p2 p3 =
   !:time. (dis t => ~(p1 t) \ ~(p2 t) \ ~(p3 t)) \ /
     (p1 t = ~(dis t) \ ~(p2 t) \ ~(p3 t)) \ /
     (p2 t = ~(dis t) \ ~(p1 t) \ ~(p3 t)) \ /
     (p3 t = ~(dis t) \ ~(p1 t) \ ~(p2 t)) \ /
     (p1 (t+1) = (p3 t)) \ /
     (p2 (t+1) = (p1 t)) \ /
     (p3 (t+1) = ((p2 t) => ~(dis (t+1)) | F))"
% there would be a race here, but it can be gotten rid of
% by feeding this block with the unstrobed "stop" right out
% of the STOP unit. It makes NO difference at the spec level %
);

%---------------------------------------------------------------------------
STOP unit
%---------------------------------------------------------------------------

let STOP_SPEC = new_definition
 ('STOP_SPEC',
   "! out inl in2 strobe.
   STOP_SPEC out inl in2 strobe =
!t:time. out (t+1) = (strobe t) => ((in1 t) \ (in2 t)) | out t

%-------------------------------------------------------------------------

MPC unit

%-------------------------------------------------------------------------

let MPC_SPEC = new_definition
('MPC_SPEC',

"! dis strobe (in:time->btT) out.
MPC_SPEC dis strobe in out =
!t:time. (out (t+1) = (strobe t) => in t |
            (dis t) => (F,F,F,F,F,F,F) |
            out t)"
);

%-------------------------------------------------------------------------

INSTRUCTION DECODER

%-------------------------------------------------------------------------

let INSDEC_SPEC = new_definition
('INSDEC_SPEC',

"! (rep:'rep_ty) (opcin:*opcode) (b enable stop reqm:bool) (opcout:btS).
INSDEC_SPEC rep opcin b enable stop opcout reqm =
(stop = (FST (decode rep opcin, b))) \ enable) /
(opcout = FST (SND (decode rep opcin, b))) /
(reqm = SND (SND (decode rep opcin, b)))"
);

%-------------------------------------------------------------------------

MICRO-SEQUENCING LOGIC (MSL)

%-------------------------------------------------------------------------

let MSL_SPEC = new_definition
('MSL_SPEC',

"! (rep:'rep_ty) (res:*_ordn) (b ovl reqm:bool) (opc:btS) (dr seqctl:bt3)
MSL_SPEC rep maddr seqctl res b ovl df mf reqm opc stop jaddr mc =
let case1 = (seqctl = (F,F,T)) in
let case2 = (seqctl = (F,T,F)) in
let case3 = (seqctl = (F,T,T)) in
let case4 = (seqctl = (T,F,F)) in
let case5 = (seqctl = (T,F,T)) in
let case6 = (seqctl = (T,T,F)) in
let case7 = (seqctl = (T,T,T)) in
let bad_res = ~(valid_address rep res) in
let pdest = ((df=(F,F,T)) \ (df=(T,F,F)) \ (df=(T,F,T))) in
  let skip = ((df=(T,F,F)) \ ~b) \ (df=(T,F,T)) \ b) in

271
let bad_rdest = (df=(T,T,F)) \ (df=(T,T,T)) in
let bad_dest = ((df=(F,T,T)) \ (df=(T,F,F)) \ (df=(T,T,F)) \ (df=(T,T,T)) in
let bad_write = (((opc=(F,F,T,T,F)) \ (opc=(F,F,T,T,T))) \ (mr=(F,F))) in

((stop = (case1 \ bad_write) \ (case4 \ ovl) \ (case5 \ bad_res) \ (case6 \ (bad_rdest \ "skip \ pdest \ bad_res")) \ (case7 \ bad_dest))) \ (jaddr = (case1 \ bad_write \ reqm) => (bt7_ival ((bt7_val maddr)+(bt2_val mf)))) !
case2 => (bt7_ival ((bt7_val maddr) + (bt5_val opc))) !
case3 => maddr !
(F,F,F,F,F,F,F))
((mc = (case1 \ reqm) \ case2 \ case3))"
);

%

ALU

%---------------------------------------------------------------

let ALU_SPEC = new_definition
('ALU_SPEC',
"! (rep:'rep_ty) (r m result:*wordn) (ovl inb outb:bool) (aluctl ff:bt4).

ALU_SPEC rep r m result ovl inb outb aluctl ff =

let case0 = (aluctl = (F,F,F,F)) in
let case1 = (aluctl = (F,F,F,T)) in
let case2 = (aluctl = (F,F,T,F)) in
let case3 = (aluctl = (F,F,T,T)) in
let case4 = (aluctl = (F,T,F,F)) in
let case5 = (aluctl = (F,T,F,T)) in
let case6 = (aluctl = (F,T,T,F)) in
let case7 = (aluctl = (F,T,T,T)) in
let case8 = (aluctl = (T,F,F,F)) in
let case9 = (aluctl = (T,F,F,T)) in
let case10 = (aluctl = (T,F,T,F)) in
let case11 = (aluctl = (T,F,T,T)) in
let case12 = (aluctl = (T,T,F,F)) in
let case13 = (aluctl = (T,T,F,T)) in
let case14 = (aluctl = (T,T,F,T)) in
let case15 = (aluctl = (T,T,T,T)) in

272
let sum = (add rep (r, m)) in
let diff = (sub rep (r, m)) in
((outb = ( case2 =~ (bcmp rep (r, m, inb, ff)) | case4 =~ (addp rep (r, m, sum)) | case6 =~ (subp rep (r, m, diff)) | case13 =~ (bit0 rep r) | case15 =~ (bitn rep r) | inb )) /
 (ovl = ( (case4 =~ case5) =~ (aovfl rep (r, m, sum)) | case6 =~ case7) =~ (sovfl rep (r, m, diff)) | case14 =~ (bitn rep r) | F )))
/
(result = ( (case0 =~ (case2)) =~ m | case1 =~ r | case3 =~ (neg rep m) | (case4 =~ case5) =~ sum | (case6 =~ case7) =~ diff | case8 =~ (bxor rep (r, m)) | case9 =~ (band rep (r, m)) | case10 =~ (bnor rep (r, m)) | case11 =~ (band rep (r, bnot rep m)) | case12 =~ (shr rep r) | case13 =~ (shr rep (r, inb)) | case14 =~ (shl rep r) | (shlb rep (r, inb)))))"
);;

%---------------------------------------------------------------------
Register block
%---------------------------------------------------------------------%

let REGISTER_BLOCK = new_definition
('REGISTER_BLOCK',
  "! (rep:'rep_ty) (regs:time->(*wordn)list) strobe din_en result_en din_sel
   addr_sel (mreg insreg result din r m:time->*wordn) (rsel msel:time->bt2)
   (result_sel mdf:time->bt3) (mar:time->*address) (ir:time->*opcode) dfc
   (b :time->bool).
   REGISTER_BLOCK rep result din strobe r_sel result_sel din_en result_en
   addr_sel din sel m sel mar ir r m regs mreg insreg dfc mdf b =
   !:time.
   ((regs (t+1) =
     (((strobe t) /
       (result_en t)) =>
       (((dfc t) /
         ((mdf t = (T,T,F)) /
           (mdf t = (T,T,T)))) =>

273
```plaintext
regs t |
( update_reg (regs t)(result_sel t)(b t)(result t)) | (regs t)) /

(mreg t+1) =
  (strobe t) =>
  ( (din_en t) => ( (din_sel t) => (mreg t) | (din t)) |
    (result_en t) \ (dfc t) \ (bt3_val(result_sel t)=6) =>
      (result t) | (mreg t)) |
    (result t) |

(ary t) /

(insreg t+1) =
  (strobe t) =>
  ( (din_en t) => ( (din_sel t) => (din t) | (insreg t)) |
    (result_en t) \ (dfc t) \ (bt3_val(result_sel t)=7) =>
      (join rep (opcode rep (insreg t), address rep (result t))) |
    (insreg t))) /

(insreg t) /

(r t =
  (EL (bt2_val (r_sel t)) (regs t)) |

(m t =
  (m_sel t) = (F,F) => (mreg t) |
  (m_sel t) = (F,T) => (wordn rep 1) |
  (pad rep (address rep (insreg t))) |

  (ir t =
    (opcode rep (insreg t))) |

  (mar t =
    (addr_sel t => (address rep (insreg t)) |
    (address rep (EL p_reg (regs t))))))

);

%---------------------------------------------------------------------
Memory
%---------------------------------------------------------------------

let EXT_INTERFACE = new_definition
('EXTINTERFACE',
"! (rep:rep_ty) rd wr io strobe addr w_data r_data ram.
EXT_INTERFACE rep rd wr io strobe addr w_data r_data ram =
!t:time .
  (ram (t+1) =
    (((wr t) \ (strobe t)) =>
      (io t => storeio rep (ram t, addr t, w_data t) |
        store rep (ram t, addr t, w_data t)) |
    ram t)) /
  (r_data t =
    (((rd t) \ (strobe t)) =>
```

let CONTROL_UNIT = new_definition
('CONTROL_UNIT',
"! (rep:'rep_ty) (mpc:time->bt7)

(ph1 ph2 ph3 stop reqm msl_stop b ovl r w io dfc de re adrs
 ds:time->bool) (rs:time->bt2) (rft mft:bt2) (ress mdf:time->bt3)
 (dft:bt3) (opc:time->bt5) (res:time->wordn) (mir:time->ucode)
 (aluctl:time->bt4) (dec_ctl :time->bool)
 (urom:(time->num->ucode)) (reset:time->bool).

CONTROL_UNIT rep mpc ph1 ph2 ph3 stop rft mft dft reqm opc msl_stop res
 b ovl mir aluctl dec_ctl r w io mdf dfc rs ress de re adrs ds

ms urom (reset) =
!

? maddr seqctl jaddr mc muxm_o mrf rfc.

(MSL_SPEC rep (maddr t) (seqctl t) (res t) (b t) (ovl t) dft mft
(reqm t) (opc t) (msl_stop t) (jaddr t) (mc t))
/

(PHASE_CLOCK_SPEC stop ph1 ph2 ph3)
/

(MUXM_SPEC (mc t)(jaddr t) (bt7_ival ((bt7_val (mpc t)) + 1)) (muxm_o t))
/

(MPC_SPEC stop ph3 muxm_o mpc)
/

(mir (t+1) = (ph1 t) => urom t (bt7_val (mpc t)) | mir t)
/

(maddr t = (Maddr (mir t))) /
(seqctl t = (Seqctl (mir t))) /
(aluctl t = (Aluctl (mir t))) /
(dec_ctl t = (Dec_ctl (mir t))) /
(r t = (R (mir t))) /
(w t = (W (mir t))) /
(io t = (Io (mir t))) /
(mrf t = (Mrf (mir t))) /
(mdf t = (Mdf (mir t))) /
(rfc t = (Rfc (mir t))) /
(df t = (Df (mir t))) /
(de t = (Dc (mir t))) /\n\
(re t = (Re (mir t))) /
(adrs t = (Adrs (mir t))) /
(ds t = (Ds (mir t))) /
(ms t = (Ms (mir t))) /
(MUXR_SPEC (rft) (rft) (rs t) /
(MUXD_SPEC (dft) (dft) (ress t))))
);

%----------------------------------------------------------
Data path
%----------------------------------------------------------%

let DATAPATH = new_definition
('DATAPATH',
"! (rep:'rep_ty) (din dout r latch m latch res mreg insreg:time->wordn)
  (b o v l r e q m s t o p m s l_st o p p h 3 w r i o d f c d i n_en r e s u l t_en
   add r_s e l d i n s e l : time->bool)
  (m a r : time->address) (o p c : time->bt5) (r e g s : time->(*wordn)list)
  (r_s e l_m_sel:time->bt2) (r f t m f t : bt2) (r e s u l t _s e l m d f : time->bt3)
  (d f t : bt3) (r a m : time->memory) (a l u c t l : time->bt4)
  (d e c_c t l r e s e t : time->bool).
DATAPATH rep din dout b mar r latch m latch res o v l opc reqm s t o p m s l_st o p
  p h 2 p h 3 r e g s mreg insreg r f t m f t r a m r d w r i o m d f d f c a l u c t l
  d e c_c t l r _ s e l r e s u l t _s e l d i n_en r e s u l t_en a d d r_s e l d i n_s e l
  m_s e l r e s e t =
  t : t i m e .
  ? d i n_i m a r_i r l a t c h_i m l a t c h_i r s u l t a l u_o v l a l u_b i r d e c_s t o p .
  ((r f t = R S F r e p (i n s r e g t)) /
  (m f t = M S F r e p (i n s r e g t)) /
  (d f t = D S F r e p (i n s r e g t)) /
  (R E G I N T E R_B L O C K r e p r e s u l t d i n p h 3 r _ s e l r e s u l t _s e l d i n_e n r e s u l t_en
    a d d r_s e l d i n_s e l m_s e l m a r_i i r r l a t c h_i m l a t c h_i r e g s mreg i n s r e g
   d f c m d f b)
 /
  (M A R_S E C (t . ((r d t) \ (w r t))) p h 2 m a r_i m a r)
 /
  (R E G_E N_S E C r d p h 3 d i n_i d i n)
 /
  (R E G_E N_S E C w r p h 2 r l a t c h_i d o u t)
 /
  (E X T_I N T E R F A C E r e p r d w r i o p h 3 m a r d o u t d i n_i r a m)
 /
  (R E G_S E C m l a t c h_i p h 2 m l a t c h)
 /
  (R E G_S E C r l a t c h_i p h 2 r l a t c h)
 /
  2 7 6
Define State and selector functions for $s:\text{time} \rightarrow \text{EBM\_state}$

\[
\text{let EBM\_state} = \\
"(:(*\text{wordn})\text{list} # \% \text{regs} % \\
\quad (*\text{wordn} # \% \text{mreg} % \\
\quad (*\text{wordn} # \% \text{insreg} % \\
\quad (*\text{wordn} # \% \text{din} % \\
\quad (*\text{wordn} # \% \text{dout} % \\
\quad (*\text{memory} # \% \text{ram} % \\
\quad (\text{bool} # \% \text{b} % \\
\quad (\text{bool} # \% \text{stop} % \\
\quad (\text{bool} # \% \text{ovl} % \\
\quad (*\text{address} # \% \text{mar} % \\
\quad (*\text{wordn} # \% \text{res} % \\
\quad (\text{bt7} # \% \text{mpc} % \\
\quad (\text{ucode} # \% \text{mir} % \\
\quad ((\text{num} \rightarrow \text{ucode}) # \% \text{urom} % \\
\quad (*\text{wordn} # \% \text{rlatch} % \\
\quad (*\text{wordn} # \% \text{mlatch} % \\
\quad (\text{bool} # \% \text{phase1} % \\
\quad (\text{bool} # \text{bool})))))())))")") ;; \% \text{phase2, phase3} %
\]

let RegsS = new\_definition

"(!((t:time) (s:time->"EBM\_state") . 
RegsS s t = FST(s t))"

let RegsS = TAC\_PROOF

"(! (t:time) (regs:time->(*\text{wordn})\text{list} 
\text{))}"

277
(mreg insreg din dout res rlatch mlatch:time->*wordn)
(ram:time->memory) (b stop ovl ph1 ph2 ph3:time->bool)
(mar:time->*address) (mpc:time->bt7) (mir:time->ucode)
(urom:num->ucode).

RegsS (\t. (regs t, mreg t, insreg t, din t, dout t, ram t, b t, stop t,
ovl t, mar t, res t, mpc t, mir t, urom, rlatch t, mlatch t,
ph1 t, ph2 t, ph3 t)) = regs",
REPEAT GEN_TAC
THEN CONV_TAC (TOP_DEPTH_CONV FUN_EQ_CONV)
THEN PURE_ONCE_REWRITE_TAC [RegsS]
THEN BETA_TAC
THEN REWRITE_TAC [];;

let MregS = new_definition
('MregS',
'! (t:time) (s:time->EBM_state).
 MregS s t = FST(SND(s t))"
);;

let MregS = TAC_PROOF
(([],
'! (t:time) (regs:time->(*wordn)list)
 (mreg insreg din dout res rlatch mlatch:time->*wordn)
 (ram:time->memory) (b stop ovl ph1 ph2 ph3:time->bool)
 (mar:time->*address) (mpc:time->bt7) (mir:time->ucode)
 (urom:num->ucode).
 MregS (\t. (regs t, mreg t, insreg t, din t, dout t, ram t, b t, stop t,
 ovl t, mar t, res t, mpc t, mir t, urom, rlatch t, mlatch t,
 ph1 t, ph2 t, ph3 t)) = mreg"),
REPEAT GEN_TAC
THEN CONV_TAC (TOP_DEPTH_CONV FUN_EQ_CONV)
THEN PURE_ONCE_REWRITE_TAC [MregS]
THEN BETA_TAC
THEN REWRITE_TAC [];;

let InsregS = new_definition
('InsregS',
'! (t:time) (s:time->EBM_state).
 InsregS s t = FST(SND(SND(s t)))"
);;

let InsregS = TAC_PROOF
(([],
'! (t:time) (regs:time->(*wordn)list)
let DinS = new_definition
(‘DinS’,
"!(t:time) (s:time->'EBM_state).
DinS s t = FST(SND(SND(SND(s t))))"
);

let DoutS = new_definition
(‘DoutS’,
"!(t:time) (s:time->'EBM_state).
DoutS s t = FST(SND(SND(SND(SND(s t))))"
);

let DoutS = TAC_PROOF
(([],
"!(t:time) (regs:time->(*wordn)list)
(mreg insreg din dout res r latch mlatch:time->*wordn)
(ram:time->*memory) (b stop ovl phl ph2 ph3:time->bool)
(mar:time->*address) (mpc:time->btT) (mir:time->ucode)
(urom:num->ucode).
DoutS (\t. (regs t, mreg t, insreg t, din t, dout t, ram t, b t, stop t,
 ovl t, mar t, res t, mpc t, mir t, urom, r latch t, mlatch t,
 ph1 t, ph2 t, ph3 t)) = dout")
);

let DoutS = TAC_PROOF
(([],
"!(t:time) (regs:time->(*wordn)list)
(mreg insreg din dout res r latch mlatch:time->*wordn)
(ram:time->*memory) (b stop ovl phl ph2 ph3:time->bool)
(mar:time->*address) (mpc:time->btT) (mir:time->ucode)
(urom:num->ucode).
DoutS (\t. (regs t, mreg t, insreg t, din t, dout t, ram t, b t, stop t,
 ovl t, mar t, res t, mpc t, mir t, urom, r latch t, mlatch t,
 ph1 t, ph2 t, ph3 t)) = dout")
);

let DoutS = TAC_PROOF
(([],
"!(t:time) (regs:time->(*wordn)list)
(mreg insreg din dout res r latch mlatch:time->*wordn)
(ram:time->*memory) (b stop ovl phl ph2 ph3:time->bool)
(mar:time->*address) (mpc:time->btT) (mir:time->ucode)
(urom:num->ucode).
DoutS (\t. (regs t, mreg t, insreg t, din t, dout t, ram t, b t, stop t,
 ovl t, mar t, res t, mpc t, mir t, urom, r latch t, mlatch t,
 ph1 t, ph2 t, ph3 t)) = dout")
)
DoutS (\t. (regs t, mreg t, insreg t, din t, dout t, ram t, b t, stop t, ovl t, mar t, res t, mpc t, mir t, urom, ratch t, mlatch t, ph1 t, ph2 t, ph3 t)) = dout),

let Rams = new_definition
('RamS',
  '!(t:time) (s:time->'EBH_state).
       Rams s t = FST(SND(SND(SND(SND(SND(s t)))))))'
);

let BS = new_definition
('BS',
  '!(t:time) (s:time->'EBM_state).
       BS s t = FST(SND(SND(SND(SND(SND(s t)))))))'
);
BS \( \langle t. (\text{regs } t, \text{mreg } t, \text{insreg } t, \text{din } t, \text{dout } t, \text{ram } t, b t, \text{stop } t, \text{ovl } t, \text{mar } t, \text{res } t, \text{mpc } t, \text{mir } t, \text{urom } t, \text{rlatch } t, \text{mlatch } t, \text{phi } t, \text{phi } t, \text{phi } t, \text{phi } t) \rangle = b\rangle),
\)
REPEAT GEN_TAC
THEN CONV_TAC (TOP_DEPTH_CONV FUN_EQ_CONV)
THEN PURE_ONCE_REWRITE_TAC [BS]
THEN BETA_TAC
THEN REWRITE_TAC []

let StopS = new_definition
('StopS',
"!(t:time) (s:time->'EBM_state).
StopS s t = FST(SND(SND(SND(SND(SND(SND(SND(s t)))))))))"
);

let StopS = TAC_PROOF
(([],
"!(t:time) (regs:time->(*worcln)list)
(mreg insreg din dout res rlatch mlatch:time->*wordn)
(ram:time->*memory) (b stop ovl phi t2 phi3:time->bool)
(mar:time->*address) (mpc:time->bt7) (mir:time->ucode)
(urom:num->ucode).
StopS \( \langle t. (\text{regs } t, \text{mreg } t, \text{insreg } t, \text{din } t, \text{dout } t, \text{ram } t, b t, \text{stop } t, \text{ovl } t, \text{mar } t, \text{res } t, \text{mpc } t, \text{mir } t, \text{urom } t, \text{rlatch } t, \text{mlatch } t, \text{phi } t, \text{phi } t, \text{phi } t) \rangle = \text{stop}\rangle),
\)
REPEAT GEN_TAC
THEN CONV_TAC (TOP_DEPTH_CONV FUN_EQ_CONV)
THEN PURE_ONCE_REWRITE_TAC [StopS]
THEN BETA_TAC
THEN REWRITE_TAC []

let OvlS = new_definition
('OvlS',
"!(t:time) (s:time->'EBM_state).
OvlS s t = FST(SND(SND(SND(SND(SND(SND(SND(s t)))))))))"
);

let OvlS = TAC_PROOF
(([],
"!(t:time) (regs:time->(*wordn)list)
(mreg insreg din dout res rlatch mlatch:time->*wordn)
(ram:time->*memory) (b stop ovl phi t2 phi3:time->bool)
(mar:time->*address) (mpc:time->bt7) (mir:time->ucode)
(urom:num->ucode).
OvlS \( \langle t. (\text{regs } t, \text{mreg } t, \text{insreg } t, \text{din } t, \text{dout } t, \text{ram } t, b t, \text{stop } t, \text{ovl } t, \text{mar } t, \text{res } t, \text{mpc } t, \text{mir } t, \text{urom } t, \text{rlatch } t, \text{mlatch } t, \text{phi } t, \text{phi } t, \text{phi } t) \rangle = \text{stop}\rangle),
\)
(mreg insreg din dout res rlatch mlatch:time->*wordn)
(ram:time->*memory) (b stop ovl ph1 ph2 ph3:time->bool)
(mar:time->*address) (mpc:time->btT) (mir:time->ucode)
(urom:num->ucode).

Ov1S (\t. (regs t, mreg t, insreg t, din t, dout t, ram t, b t, stop t,
  ovl t, mar t, res t, mpc t, mir t, urom, rlatch t, mlatch t,
  ph1 t, ph2 t, ph3 t)) = ov1"),

REPEAT GEN_TAC
THEN CONV_TAC (TOP_DEPTH_CONV FUN_EQ_CONV)
THEN PURE_ONCE_REWRITE_TAC [Ov1S]
THEN BETA_TAC
THEN REWRITE_TAC []
);

let MarS = new_definition
('MarS',
"!(t:time) (s:time->'EBM_state).
  MarS s t = PST(SND(SND(SND(SND(SND(SND(SND(SND(SND(s t)))))))))))"
);

let MarS = TAC_PROOF
(([]),
"!(t:time) (regs:time->(*ordm)list)
  (mreg insreg din dout res rlatch mlatch:time->*wordn)
  (ram:time->*memory) (b stop ovl ph1 ph2 ph3:time->bool)
  (mar:time->*address) (mpc:time->btT) (mir:time->ucode)
  (urom:num->ucode).

  MarS (\t. (regs t, mreg t, insreg t, din t, dout t, ram t, b t, stop t,
    ovl t, mar t, res t, mpc t, mir t, urom, rlatch t, mlatch t,
    ph1 t, ph2 t, ph3 t)) = mar"),

REPEAT GEN_TAC
THEN CONV_TAC (TOP_DEPTH_CONV FUN_EQ_CONV)
THEN PURE_ONCE_REWRITE_TAC [MarS]
THEN BETA_TAC
THEN REWRITE_TAC []
);

let ResS = new_definition
('ResS',
"!(t:time) (s:time->'EBM_state).
  ResS s t = FST(SND(SND(SND(SND(SND(SND(SND(SND(SND(s t)))))))))))"
);

let ResS = TAC_PROOF
(([]),
"!(t:time) (regs:time->(*ordm)list)

282
(mreg insreg din dout res rlap tch mlatch:time->*wordn)
(ram:time->*memory) (b stop ovl ph1 ph2 ph3:time->bool)
(mar:time->*address) (mpc:time->bt7) (mir:time->ucode)
(urom:um->ucode).

resS (t. (regs t, mreg t, insreg t, din t, dou t, ram t, b t, stop t,
ovl t, mar t, res t, mpc t, mir t, urom, rlatch t, mlatch t,
phi t, ph2 t, ph3 t)) = res",

REPEAT GEN_TAC
THEN CONV_TAC (TOP_DEPTH_CONV FUN_EQ_CONV)
THEN PURE_ONCE_REWRITE_TAC [resS]
THEN BETA_TAC
THEN REWRITE_TAC []

let mpcS = new_definition
('mpcS',
"! (t:time) (s:time->'EBM_state).
mpcS s t = FST(SND(SND(SND(SND(SND(SND(SND(SND(SND(s t))))))))))")

let mpcS = TAC_PROOF
(([]),
"! (t:time) (regs:time->(*wordn)list)
(mreg insreg din dout res rlatch mlatch:time->*wordn)
(ram:time->*memory) (b stop ovl ph1 ph2 ph3:time->bool)
(mar:time->*address) (mpc:time->bt7) (mir:time->ucode)
(urom:um->ucode).
mpcS (t. (regs t, mreg t, insreg t, din t, dou t, ram t, b t, stop t,
ovl t, mar t, res t, mpc t, mir t, urom, rlatch t, mlatch t,
phi t, ph2 t, ph3 t)) = mpc")

REPEAT GEN_TAC
THEN CONV_TAC (TOP_DEPTH_CONV FUN_EQ_CONV)
THEN PURE_ONCE_REWRITE_TAC [mpcS]
THEN BETA_TAC
THEN REWRITE_TAC []

let mirS = new_definition
('mirS',
"! (t:time) (s:time->'EBM_state).
mirS s t = FST(SND(SND(SND(SND(SND(SND(SND(SND(SND(s t))))))))))")

let mirS = TAC_PROOF
(([]),
"! (t:time) (regs:time->(*wordn)list)
let MirS = new_definition
('MirS',
"!(t:time) (s:time->'EBM_state).
MirS s t = mir")
);

let UromS = new_definition
('UromS',
"!(t:time) (s:time->'EBM_state).
UromS s t = urom")
);

let RlatchS = new_definition
('RlatchS',
"!(t:time) (s:time->'EBM_state).
RlatchS s t = rlatch")
);
(mreg insreg din dout res rlatch mlatch:time->*wordn)
(ram:time->*memory) (b stop ovl phl ph2 ph3:time->bool)
(mar:time->*address) (mpc:time->bt7) (mir:time->ucode)
(urom:num->ucode).

RlatchS (
\(t. (\text{regs } t, \text{mreg } t, \text{insreg } t, \text{din } t, \text{dout } t, \text{ram } t, \text{b } t, \text{stop } t,
\text{ovl } t, \text{mar } t, \text{res } t, \text{mpc } t, \text{mir } t, \text{urom}, \text{rlatch } t, \text{mlatch } t,
\text{phl } t, \text{ph2 } t, \text{ph3 } t)) = \text{rlatch}
),

REPEAT GEN_TAC
THEN CONV_TAC (TOP_DEPTH_CONV FUN_EQ_CONV)
THEN PURE_ONCE_REWRITE_TAC [RlatchS]
THEN BETATAC
THEN REWRITE_TAC []
);

let MlatchS = new_definition
('MlatchS',
"!(t:time) (s:time->'EBM_state).
MlatchS s t = FST(SND(SND(SND(SND(SND(SND(SND(SND(SND(SND(SND(SND(SND(SND(SND(s t))))))))))))))))")
);

let MlatchS = TAC_PROOF
(([],
"!(t:time) (regs:time->(*wordn)list)
\(t. (\text{mreg } t, \text{insreg } t, \text{din } t, \text{dout } t, \text{ram } t, \text{b } t, \text{stop } t,
\text{ovl } t, \text{mar } t, \text{res } t, \text{mpc } t, \text{mir } t, \text{urom}, \text{rlatch } t, \text{mlatch } t,
\text{phl } t, \text{ph2 } t, \text{ph3 } t)) = \text{mlatch}
),

REPEAT GEN_TAC
THEN CONV_TAC (TOP_DEPTH_CONV FUN_EQ_CONV)
THEN PURE_ONCE_REWRITE_TAC [MlatchS]
THEN BETATAC
THEN REWRITE_TAC []
);

let PhlS = new_definition
('PhlS',
"!(t:time) (s:time->'EBM_state).
PhlS s t = FST(SND(SND(SND(SND(SND(SND(SND(SND(SND(SND(SND(SND(SND(SND(SND(s t))))))))))))))))")
);
let Ph1S = TAC_PROOF
(([], "!(t:time) (regs:time->(*wordn)list)
 (mreg insreg din dout res rlatch mlatch:time->*wordn)
 (ram:time->*memory) (b stop ovl ph1 ph2 ph3:time->bool)
 (mar:time->*address) (mpc:time->bt7) (mir:time->ucode)
 (urom:num->ucode).
 Ph1S (\t. (regs t, mreg t, insreg t, din t, dout t, ram t, b t, stop t, ovl t, mar t, res t, mpc t, mir t, urom, rlatch t, mlatch t, ph1 t, ph2 t, ph3 t)) = ph1"),
 REPEAT GEN_TAC
 THEN CONV_TAC (TOP_DEPTH_CONV FUN_EQ_CONV)
 THEN PURE_ONCE_REWRITE_TAC [Ph1S]
 THEN BETA_TAC
 THEN REWRITE_TAC []);

let Ph2S = new_definition
('Ph2S', "!(t:time) (s:time->'EBM_state).
 Ph2S s t = FST(SND(SND(SND(SND(SND(SND(SND(SND(s t))))))))))")
);

let Ph2S = TAC_PROOF
(([], "!(t:time) (regs:time->(*wordn)list)
 (mreg insreg din dout res rlatch mlatch:time->*wordn)
 (ram:time->*memory) (b stop ovl ph1 ph2 ph3:time->bool)
 (mar:time->*address) (mpc:time->bt7) (mir:time->ucode)
 (urom:num->ucode).
 Ph2S (\t. (regs t, mreg t, insreg t, din t, dout t, ram t, b t, stop t, ovl t, mar t, res t, mpc t, mir t, urom, rlatch t, mlatch t, ph1 t, ph2 t, ph3 t)) = ph2"),
 REPEAT GEN_TAC
 THEN CONV_TAC (TOP_DEPTH_CONV FUN_EQ_CONV)
 THEN PURE_ONCE_REWRITE_TAC [Ph2S]
 THEN BETA_TAC
 THEN REWRITE_TAC []);

let Ph3S = new_definition
('Ph3S', "!(t:time) (s:time->'EBM_state).

286
let Ph3S = TAC_PROOF
(([],
  "! (t:time) (regs:time->wordnlist)
  (mreg insreg din dout res rlatch mlatch:time->wordn)
  (ram:time->memory) (b stop ovl ph1 ph2 ph3:time->bool)
  (mar:time->address) (mpc:time->bt7) (mir:time->ucode)
  (urom:num->ucode).
  Ph3S (\t. (regs t, mreg t, insreg t, din t, dout t, ram t, b t, stop t,
    ovl t, mar t, res t, mpc t, mir t, urom, rlatch t, mlatch t,
    ph1 t, ph2 t, ph3 t)) = ph3”),
REPEAT GEN_TAC
THEN CONV_TAC (TOP_DEPTH_CONV FUN_EQ_CONV)
THEN PURE_ONCE_REWRITE_TAC [Ph3S]
THEN BETA_TAC
THEN REWRITE_TAC []);

let EBM_env = "$:bool$"

let ResetE = new_definition
('ResetE',
  "! (t:time) (e:time->EBM_env) .
  ResetE e t = e t"
);

let ResetE = TAC_PROOF
(([],
  "! (t:time) (reset:time->bool) .
  ResetE (\t. reset t) = reset”)
REPEAT GEN_TAC
THEN CONV_TAC (TOP_DEPTH_CONV FUN_EQ_CONV)
THEN PURE_ONCE_REWRITE_TAC [ResetE]
THEN BETA_TAC
THEN REWRITE_TAC []);

%------------------------------------------------------
Define Electronic Block Model
%------------------------------------------------------

let EBM_def = new_definition
('EBM_def',
  "! (rep:'rep_ty) (s:time->EBM_state) (e:time->EBM_env) .
  
  Ph3S s t = SND(SND(SND(SND(SND(SND(SND(SND(SND(SND(SND(SND(SND(SND(SND(SND(s t))))))))))))))))")

%------------------------------------------------------
EBM rep s e =
    ? opc reqm msl_stop rf mf df rd wr io mdf dfc aluctl dec_ctl
    r_sel result Sel din_en result_en addr_sel dinSel m_sel.
    (DATAPATH rep (Dins s) (DoutS s) (BS s) (MarS s) (RlatchS s)
    (MlatchS s) (ResS s) (OvlS s) opc reqm (StopS s) msl_stop
    (Ph2S s) (Ph3S s) (RegS s) (MregS s) (InsregS s) rf mf df (RamS s)
    rd wr io mdf dfc aluctl dec_ctl r_sel result Sel din_en
    result_en addr_sel dinSel m_sel (ResetE e)) /
    (CONTROL_UNIT rep (MpcS s) (Ph1S s) (Ph2S s) (Ph3S s) (StopS s) rf mf
    df reqm opc msl_stop (ResS s) (BS s) (OvlS s) (MregS s) aluctl dec_ctl
    rd wr io mdf dfc r_sel result Sel din_en result_en addr_sel dinSel m_sel
    (UromS s) (ResetE e))"
);

let EBM = prove_thm
('EBM',
  "!(rep:*rep_ty) (regs:time->(*wordn)list)
    (mreg insreg din dout:time->*wordn) (ram:time->*memory)
    (r stop ovl:time->bool) (mar:time->*address) (res:time->*wordn)
    (mpc:time->btT) (mir:time->ucode) (urom:num->ucode)
    (rlatch mlatch:time->*wordn) (ph1 ph2 ph3:time->bool)
    (reset:time->bool).
EBM rep (\ t. (regs t, mreg t, insreg t, din t, dout t, ram t, b t, stop t,
    ovl t, mar t, res t, mpc t, mir t, urom, rlatch t, mlatch t,
    ph1 t, ph2 t, ph3 t))
(\ t. (reset t)) =
    ? opc reqm msl_stop rf mf df rd wr io mdf dfc aluctl dec_ctl
    r_sel result Sel din_en result_en addr_sel dinSel m_sel.
    (DATAPATH rep din dout b mar rlatch mlatch res ovl opc reqm stop
    msl_stop ph2 ph3 regs mreg insreg rf mf df ram rd wr io mdf dfc
    aluctl dec_ctl r_sel result Sel din_en result_en addr_sel
    dinSel m_sel reset) /
    (CONTROL_UNIT rep mpc ph1 ph2 ph3 stop rf' mf df' reqm
    opc msl_stop res b ovl mir aluctl dec_ctl rd wr io mdf dfc r_sel
    result Sel din_en result_en addr_sel dinSel m_sel ((\ t:time. urom)
    reset))",
REWRITE_TAC [RegsS; MregS; InsregS; DinS; DoutS; RamS; BS; StopS;
    OvlS; MarS; ResS; MpcS; MirS; UromS; RlatchS; MlatchS;
    Ph1S; Ph2S; Ph3S; ResetE; EBM_def]
);

let EBM_expanded = save_thm
('EBM_expanded',
(CONV_RULE (TOP_DEPTH_CONV BETA_CONV)

288
Define a function that maps EBM state to the EBM counter.

let GetEBMClock = new_definition
('GetEBMClock',
 "! (regs:*wordn)list (mreg insreg din dout:*wordn) (ram:*memory)
 (b stop ovl:bool) (mar:*address) (res:*wordn) (mpc:bt7) (mir:ucode)
 GetEBMClock (regs, mreg, insreg, din, dout, ram, b, stop, ovl, mar, res,
 mpc, mir, urom, rlatch, mlatch, phi, ph2, ph3) (reset) =
 0x:bool.F"
);

let EBM_Start = new_definition
('EBM_Start',
 "EBM_Start = 0x:bool.F"
);

close_theory();
Appendix J: INSTRUCTION DECODER

CASES FOR THE DECODER:

1. bcmp
   CSF

2. writeio
   "CSF \ DSF = (T,T,T)

3. writem
   "CSF \ DSF = (T,T,F)

4. noop
   "CSF /
   "(DSF = (T, T, T) \ DSF = (T, T, F)) /
   (DSF = (T, F, F) \ "b)

5.noop
   "CSF /
   "(DSF = (T, T, T) \ DSF = (T, T, F)) /
   (DSF = (T, F, T) \ b)

6. call
   "CSF /
   "(DSF = (T, T, T) \ DSF = (T, T, F)) /
   "((DSF = (T, F, T) \ "b) \ (DSF = (T, F, F) \ b))
   FSF = (F, F, F, T)

7. neg
   "CSF /
   "(DSF = (T, T, T) \ DSF = (T, T, F)) /
   "((DSF = (T, F, T) \ "b) \ (DSF = (T, F, F) \ b))
   FSF = (F, F, F, T)
   FSF = (F, F, F, F)

8. readio
   "CSF /
   "(DSF = (T, T, T) \ DSF = (T, T, F)) /
   "((DSF = (T, F, T) \ "b) \ (DSF = (T, F, F) \ b))
   FSF = (F, F, F, T)
   FSF = (F, F, T, F)
9. readm

```
¬CSF ∨
¬(DSF = (T, T, T) ∨ DSF = (T, T, F)) ∨
¬((DSF = (T, F, T) ∧ ¬b) ∨ (DSF = (T, F, F) ∨ b))
¬FSF = (F, F, F, T) ∨
FSF = (F, F, T, T)
```

10. addb

```
¬CSF ∨
¬(DSF = (T, T, T) ∨ DSF = (T, T, F)) ∨
¬((DSF = (T, F, T) ∧ ¬b) ∨ (DSF = (T, F, F) ∨ b))
¬FSF = (F, F, F, T) ∨
FSF = (F, F, T, F)
```

11. adds

```
¬CSF ∨
¬(DSF = (T, T, T) ∨ DSF = (T, T, F)) ∨
¬((DSF = (T, F, T) ∧ ¬b) ∨ (DSF = (T, F, F) ∨ b))
¬FSF = (F, F, F, T) ∨
FSF = (F, T, F, T)
```

12. subb

```
¬CSF ∨
¬(DSF = (T, T, T) ∨ DSF = (T, T, F)) ∨
¬((DSF = (T, F, T) ∧ ¬b) ∨ (DSF = (T, F, F) ∨ b))
¬FSF = (F, F, F, T) ∨
FSF = (F, T, T, F)
```

13. subo

```
¬CSF ∨
¬(DSF = (T, T, T) ∨ DSF = (T, T, F)) ∨
¬((DSF = (T, F, T) ∧ ¬b) ∨ (DSF = (T, F, F) ∨ b))
¬FSF = (F, F, F, T) ∨
FSF = (F, T, T, T)
```

14. xor

```
¬CSF ∨
¬(DSF = (T, T, T) ∨ DSF = (T, T, F)) ∨
¬((DSF = (T, F, T) ∧ ¬b) ∨ (DSF = (T, F, F) ∨ b))
¬FSF = (F, F, F, T) ∨
FSF = (F, T, T, T)
```

15. and

```
¬CSF ∨
¬(DSF = (T, T, T) ∨ DSF = (T, T, F)) ∨
```

292
\texttt{\textasciitilde((DSF = (T, F, T) \& \textasciitilde b) \lor (DSF = (T, F, F) \& b))}
\texttt{FSF = (F, F, F, T) \lor}
\texttt{FSF = (T, F, F, T)}

\texttt{16. \texttt{nor}}
\texttt{\textasciitilde CSF \lor}
\texttt{\textasciitilde((DSF = (T, T, T) \lor DSF = (T, T, F)) \lor}
\texttt{\textasciitilde((DSF = (T, F, T) \& \textasciitilde b) \lor (DSF = (T, F, F) \& b))}
\texttt{FSF = (F, F, F, T) \lor}
\texttt{FSF = (T, F, F, T)}

\texttt{17. \texttt{andmbar}}
\texttt{\textasciitilde CSF \lor}
\texttt{\textasciitilde((DSF = (T, T, T) \lor DSF = (T, T, F)) \lor}
\texttt{\textasciitilde((DSF = (T, F, T) \& \textasciitilde b) \lor (DSF = (T, F, F) \& b))}
\texttt{FSF = (F, F, F, T) \lor}
\texttt{FSF = (T, F, T, T)}

\texttt{18. \texttt{shr}}
\texttt{\textasciitilde CSF \lor}
\texttt{\textasciitilde((DSF = (T, T, T) \lor DSF = (T, T, F)) \lor}
\texttt{\textasciitilde((DSF = (T, F, T) \& \textasciitilde b) \lor (DSF = (T, F, F) \& b))}
\texttt{FSF = (F, F, F, T) \lor}
\texttt{FSF = (T, T, F, T) \lor MSF = (F, F)}

\texttt{19. \texttt{shr2}}
\texttt{\textasciitilde CSF \lor}
\texttt{\textasciitilde((DSF = (T, T, T) \lor DSF = (T, T, F)) \lor}
\texttt{\textasciitilde((DSF = (T, F, T) \& \textasciitilde b) \lor (DSF = (T, F, F) \& b))}
\texttt{FSF = (F, F, F, T) \lor}
\texttt{FSF = (T, T, F, F) \lor MSF = (F, T)}

\texttt{20. \texttt{shl}}
\texttt{\textasciitilde CSF \lor}
\texttt{\textasciitilde((DSF = (T, T, T) \lor DSF = (T, T, F)) \lor}
\texttt{\textasciitilde((DSF = (T, F, T) \& \textasciitilde b) \lor (DSF = (T, F, F) \& b))}
\texttt{FSF = (F, F, F, T) \lor}
\texttt{FSF = (T, T, F, F) \lor MSF = (T, F)}

\texttt{21. \texttt{shlb}}
\texttt{\textasciitilde CSF \lor}
\texttt{\textasciitilde((DSF = (T, T, T) \lor DSF = (T, T, F)) \lor}
\texttt{\textasciitilde((DSF = (T, F, T) \& \textasciitilde b) \lor (DSF = (T, F, F) \& b))}
\texttt{FSF = (F, F, F, T) \lor}
\texttt{FSF = (T, T, F, F) \lor MSF = (T, T)}

\texttt{22. \texttt{error}}
"CSF /

"(DSF = (T, T, T) \ DSF = (T, T, F)) /

"((DSF = (T, F, T) \ ~b) \ DSF = (T, F, F) \ b))

"FSF = (F, F, F, T) /

FSF = (T, T, T, F)

23. error

"CSF /

"(DSF = (T, T, T) \ DSF = (T, T, F)) /

"((DSF = (T, F, T) \ ~b) \ DSF = (T, F, F) \ b))

"FSF = (F, F, F, T) /

FSF = (T, T, T, F)

24. error

"CSF /

"(DSF = (T, T, T) \ DSF = (T, T, F)) /

"((DSF = (T, F, T) \ ~b) \ DSF = (T, F, F) \ b))

"FSF = (F, F, F, T) /

FSF = (T, T, T, T)
Formal Verification of a Microcoded VIPER Microprocessor Using HOL

Karl Levitt  Sara Kalvala  Mark Heckman
Tej Kumar Arora  E. Thomas Schubert  Gerald C. Cohen
Tony Leung  Philip Windley

Boeing Defense and Space Group
Military Airplanes Division
P.O. Box 3707, M/S 4C-70
Seattle, WA 98124-2207

Levitt, Arora, Leung, Kalvala, Schubert, Windley, and Heckman: University of California, Davis, CA; Cohen: Boeing Defense & Space Group, Seattle, WA. Langley Technical Monitor: Sally C. Johnson

RSRE and members of the Hardware Verification Group at Cambridge University conducted a joint effort to prove the correspondence between the electronic block model and the top level specification of Viper. Unfortunately, the proof became too complex and unmanageable within the given time and funding constraints, and is thus incomplete as of the date of this report.

This report describes an independent attempt to use the HOL mechanical verifier to verify Viper. Deriving from recent results in hardware verification research at UC Davis, the approach has been to redesign the electronic block model to make it microcoded and to structure the proof in a series of increasingly abstract interpreter levels, the lowest being the electronic block level. The highest level is the RSRE Viper instruction set. Owing to the new approach and some results on the proof of generic interpreters as applied to simple microprocessors, this attempt required an effort approximately an order of magnitude less than the previous one.