Analysis of Compiled Code: A Prototype Formal Model

R.D. Arthan

Lemma 1 Ltd.
2nd Floor, 31A Chain Street,
Reading UK RG1 2HX
rda@lemma-one.com

Abstract. This paper reports on an experimental application of formal specification to inform analysis of compiled code. The analyses with which we are concerned attempt to recover abstraction and order from the potentially chaotic world of machine code. To illustrate the kind of abstractions of interest, we give a formal model of a simple microprocessor. This is based on a traditional state-based Z specification, but builds on that to produce a behavioural model of the microprocessor. We use the behavioural model to specify a higher-order notion: the concept of a program whose control flow can be decomposed into basic blocks. Finally, we report on the use of our techniques in the development of tools for analysis of compiled code for a real microprocessor.

1 Introduction

1.1 Background

Much of the emphasis in formal methods research has been into formalisation of the process of specifying and designing systems. However techniques and tools for analysing software engineering artefacts are of considerable importance. This paper is intended to give a simple example of how notations such as Z may be used to provide rigorous foundations for program analysis.

Because of its importance in safety-critical applications (such as avionics systems), we are particularly concerned with analysis of compiled code. Rigorous development methods greatly increase confidence in the outputs of the systems engineering process. However, those outputs still require validation, typically by inspection and testing. We believe that automated or semi-automated analysis of compiled code will be of increasing importance for system validation and that it deserves a rigorous mathematical foundation.

1.2 Program Analysis

The formal model that we present as an example of our approach is adapted from a specification originally written in late 1997. At that time, Dr. Tom Lake of Interglossa and the present author were doing some preliminary work on techniques for analysing and verifying compiled code.

Our thinking was influenced by Interglossa's suite of reverse engineering tools called REAP. These tools carry out static analysis on assembly language code to retrofit the kind of abstractions that one might expect to find in compiler-generated code. Programs that pass the conditions imposed by the analysis conform to disciplines of control flow and data access of the sort typically obeyed by a compiler.

In REAP, the analysis provides a semantic justification for a translation of the assembly language code into a higher level language such as C. Our belief was (and remains) that this kind of analysis should also justify a tractable approach to formal reasoning about low level code.

1.3 Formal Model

To provide a formal underpinning of the kinds of analysis we have in mind requires a formal model of the execution environment for the code being analysed. Towards this goal, the present author wrote a behavioural model of a simple but general microprocessor. The idea was that the control flow and data access disciplines of interest can be formally defined as constraints on the possible behaviours of particular programs.

Not all programs will conform to the disciplines that we impose; however, those that do should be significantly more amenable to formal reasoning. In safety-critical applications, we would claim that non-conforming programs should be deemed unacceptable. This would be the analogue at the machine code level of the use of "safe" high-level language subsets (such as SPARK-Ada). One would expect compilers to generate conforming target code for most reasonable high level language programs.

To demonstrate a simple form of control flow and data access discipline, we formalise the notion of a basic blocks decomposition of a program running on the microprocessor. If a basic blocks decomposition can be shown to be correct, then we know that the program is not self-modifying and never executes data. In other words, we can validly treat the program as running on a Harvard architecture rather than the von Neumann architecture of the physical microprocessor.

1.4 Expressing Higher-Order Properties in Z

The basic blocks abstraction is a so-called higher-order property; i.e., it cannot be expressed as a constraint on one state transition, but rather has to be expressed in terms of complete execution histories. Traditional methods for using Z focus on specification of a system as a state-transition machine, i.e., via first-order properties alone. However, Z provides all the mathematical features needed to specify higher-order properties and the schema calculus helps to abbreviate many of the definitions. The approach is to construct a behavioural model of the system in terms of a specification in the traditional style. This paper is intended both to illustrate and to promote this approach to formal modelling.

1.5 Specification of Program Analyses

Many algorithms have the characteristic that it is much easier to specify the solution to be found than it is to design and specify an algorithm that finds it. Milner's type inference for ML, [6] is an example. Discovering properties like the basic blocks abstraction by automatic analysis often involves techniques such as abstract interpretation which are algorithmically complex.

We believe it is important to have rigorous specifications of what the results of such analyses mean. The specification of the basic blocks abstraction in this paper is intended to demonstrate that it is possible to give rigorous and concise definitions of the requirements on a program analysis without giving the implementation detail.

1.6 Structure of the Paper

The rest of this paper is organised as follows:

- Section 2 gives a model of the simple microprocessor. This is a behavioural model, i.e., it characterises a program running on the microprocessor by its input/output relation. As an aside, we show how the behavioural model allows us to formalise the concept of refinement.
- Section 3 specifies the notion of a decomposition of a program into basic blocks. This demonstrates a simple, but not untypical, example of the kind of property that advanced program analysis techniques are used to find.
- Section 4 gives some concluding remarks including a list of the shortcomings of the simple model we present here and a discussion of how some of these were addressed in a real-life example. An index to the Z specification is given at the end of this section.

2 Processor Model

In this section we give a complete behavioural model of a somewhat idealised microprocessor. In a real example, we would have rather more work to do transcribing the manufacturer's data sheets along the lines of the Z framework we set up here. However, the work has not been found to be excessive on an actual example.

Our approach is first to develop a state-transition model of the microprocessor using the traditional Z style for specifying a sequential system [8, 9]. This is not dissimilar in spirit to the specification in chapter 9 of [3], although we choose to abstract away some of the details such as instruction decoding. We then use the state-transition model to construct a *behavioural model* — a specification of the observable behaviour of the microprocessor formulated as a relation between input and output streams. In more detail, the plan of the specification is as follows:

- First of all, in section 2.1, we give our model of the registers and memory of the microprocessor. These provide "data types" that are used throughout the rest of the specification.
- In section 2.2, we define the state space of the microprocessor.
- In section 2.3, we describe a kind of abstract syntax for the instruction set of the microprocessor.
- In section 2.4, we specify in the traditional Z style how the instructions of section 2.3 are executed.
- In section 2.5 we pull together the operations defined in section 2.4 into a single schema, $TICK$, describing one processor execution cycle.
- Finally, in section 2.6 we define sets to represent input and output streams and use the schema $TICK$ to construct the behavioural model.

The specification is written in the dialect of Z supported by the ProofPower system, [2, 5], which was used to prepare and type-check all the Z in this document. The global variables in the Z are listed in the index in section 4 and are shown in a ***bold*** font at the point of their definition.

2.1 Register and Memory Model

The following otherwise unspecified positive numbers give the maximum values of a memory word and of a memory address.

$$\boldsymbol{MAX\_WORD}, \boldsymbol{MAX\_ADDR} : \mathbb{N}_1$$

The following sets give the types for *words* in memory and memory *addresses*:

$$\boldsymbol{WORD} \mathrel{\widehat{=}} 0 \mathrel{..} MAX\_WORD$$

$$\boldsymbol{ADDR} \mathrel{\widehat{=}} 0 \mathrel{..} MAX\_ADDR$$

There is a set of *registers*, possibly empty. Some of these may be memory-mapped. We can identify memory-mapped registers by their addresses and other registers by numbers outside the address space.

$$\boldsymbol{REGISTER} : \mathbb{F}\ \mathbb{Z}$$

A storage *location* is then either a register or an address (or possibly both).

$$\boldsymbol{LOCATION} \mathrel{\widehat{=}} REGISTER \cup ADDR$$

The *store* is a total function mapping storage locations (either register identifiers or memory addresses) to words:

$$\boldsymbol{STORE} \mathrel{\widehat{=}} LOCATION \rightarrow WORD$$

Some of the memory may be *ROM*. The set of ROM addresses is some subset of the address space:

$$\boldsymbol{ROM} : \mathbb{P}\ ADDR$$

Some of the store (memory or registers) may be given over to memory-mapped I/O. The following two sets are the locations which serve as *input and output ports*. They may overlap with each other but not with the ROM:

$$\boldsymbol{IN\_PORTS}, \boldsymbol{OUT\_PORTS} : \mathbb{P}\ LOCATION$$

$$IN\_PORTS \cap ROM = OUT\_PORTS \cap ROM = \varnothing$$

2.2 Processor State

The processor has one special register: the program counter, PC. For simplicity, we model the contents of the program counter as a component of the processor state in its own right, rather than assigning a location in the store for it, as we do for other registers. This simplification does mean that the program counter cannot be memory-mapped, but that is appropriate for the vast majority of modern microprocessor types.

The *processor state* is thus given by the following schema:

PROCESSOR_STATE

$$pc : ADDR;$$
$$store : STORE$$

2.3 Instruction Set

We will give a "syntax" for instructions which actually embeds most of the semantics of computational and test instructions. A *computation* is any total function on processor states delivering a result comprising a location and a word; the location indicates where the word is to be stored.

$$\boldsymbol{COMPUTATION} \mathrel{\widehat=} PROCESSOR\_STATE \rightarrow (LOCATION \times WORD)$$

A *test* is any set of words: a word w satisfies test t iff. $w \in t$.

$$\boldsymbol{TEST} \mathrel{\widehat=} \mathbb{P}\ WORD$$

Informally, the syntax and semantics of the instruction set is as shown in the following table:

Instruction	Operands	Description
Compute	comp	Perform computation comp giving a pair (l, w); Store w at location l.
StorePC	loc	Store the contents of PC at loc.
Jump	addr	Assign addr to PC.
CondJump	loc, test, addr_1, addr_2	If the contexts of location loc satisfy test, then assign addr_1 to PC, otherwise assign addr_2 to PC.
LoadPC	loc	Assign the contents of loc to PC.

For simplicity, we specify that if any instruction attempts to write to the ROM, then the write is ignored. This aspect of the model would need to be reconciled with the actual behaviour of a particular microprocessor configuration in a real world example.

The conditional jump instruction, CondJump, is unlike most, if not all, real microprocessors in having an "if-then-else" effect, rather than "if-then". This is technically convenient in the sequel and simply requires us to encode the usual "if-then" behaviour using an "else-branch" which jumps to the instruction immediately after the conditional jump.

The StorePC and LoadPC instructions would be used by a compiler to implement subroutine call and return. Most real microprocessors offer a combination of StorePC and some kind of jump instruction as a single "jump-to-subroutine" or "call" instruction.

The instruction set is modelled by the following Z free type[1]

$$
\begin{array}{lll}
\boldsymbol{INSTRUCTION} & ::= & \\
 & & \boldsymbol{Compute}\ (COMPUTATION) \\
 & | & \boldsymbol{StorePC}\ (LOCATION) \\
 & | & \boldsymbol{Jump}\ (ADDR) \\
 & | & \boldsymbol{LoadPC}\ (LOCATION) \\
 & | & \boldsymbol{CondJump}\ (LOCATION \times TEST \times ADDR \times ADDR)
\end{array}
$$

2.4 Instruction Execution

We now describe the state change caused by execution of a single instruction using an operation schema for each type of instruction. These operations have an input parameter which is the instruction being executed. Each operation only fires if the parameter is the instruction dealt with by that operation.

A Compute instruction is executed by carrying out the computation in the current state to give a location-word pair (l, w) then updating the store by

[1] The ProofPower dialect of Z does not currently support the chevron brackets around the sets in the branches of a free type required in other Z dialects.

writing w to location l, provided l is not in ROM. If l is in ROM, then by the assumptions made in section 2.3, the store is unchanged[2].

$$
\begin{array}{|l}
\textbf{\textit{COMPUTE}} \\
\hline
\quad instr? : INSTRUCTION; \\
\quad \Delta PROCESSOR\_STATE \\
\hline
\exists\ comp : COMPUTATION;\ l : LOCATION;\ w : WORD \\
|\quad instr? = Compute\ comp \\
\bullet\quad (l,\ w) = comp\ (\theta PROCESSOR\_STATE) \\
\wedge\quad pc' = (pc + 1)\ mod\ MAX\_ADDR \\
\wedge\quad (l \notin ROM \wedge store' = store \oplus \{l \mapsto w\} \vee l \in ROM \wedge store' = store) \\
\hline
\end{array}
$$

A StorePC instruction causes the current value of the program counter to be written to the store in the location given by the operand of the instruction. The rule about attempts to write to ROM is the same as for the Compute instructions.

$$
\begin{array}{|l}
\textbf{\textit{STORE\_PC}} \\
\hline
\quad instr? : INSTRUCTION; \\
\quad \Delta PROCESSOR\_STATE \\
\hline
\exists\ l : LOCATION \mid instr? = StorePC\ l \\
\bullet\quad pc' = (pc + 1)\ mod\ MAX\_ADDR \\
\wedge\quad (l \notin ROM \wedge store' = store \oplus \{l \mapsto pc\} \vee l \in ROM \wedge store' = store) \\
\hline
\end{array}
$$

Jump A Jump instruction assigns a new value to the program counter, which will cause a transfer of control on the next execution cycle.

$$
\begin{array}{|l}
\textbf{\textit{JUMP}} \\
\hline
\quad instr? : INSTRUCTION; \\
\quad \Delta PROCESSOR\_STATE \\
\hline
\exists\ a : ADDR \mid instr? = Jump\ a \bullet pc' = a \wedge store' = store \\
\hline
\end{array}
$$

[2] In many typical microprocessor configurations, attempting to write to ROM gives "undefined" behaviour, this can readily be modelled by removing the disjunct $l \in ROM \wedge store' = store$ from the predicates of $COMPUTE$ and $STORE\_PC$.

A CondJump instruction evaluates the test and assigns a new value to the program counter according to the result of the test. This will cause the required transfer of control on the next execution cycle.

$$
\begin{array}{l}
\textbf{\textit{COND\_JUMP}} \\
\hline
instr? : INSTRUCTION; \\
\Delta PROCESSOR\_STATE \\
\hline
\exists\, l : LOCATION;\ t : TEST;\ a_1,\ a_2 : ADDR \\
\mid\quad instr? = CondJump\ (l,\ t,\ a_1,\ a_2) \\
\bullet\quad (store\ l \in t \land pc' = a_1 \lor store\ l \notin t \land pc' = a_2) \land store' = store \\
\hline
\end{array}
$$

A LoadPC instruction assigns the contents of the indicated store location to the program counter, which will cause a transfer of control on the next execution cycle.

$$
\begin{array}{l}
\textbf{\textit{LOAD\_PC}} \\
\hline
instr? : INSTRUCTION; \\
\Delta PROCESSOR\_STATE \\
\hline
\exists\, l : LOCATION \mid instr? = LoadPC\ l \bullet pc' = store\ l \land store' = store \\
\hline
\end{array}
$$

2.5 CPU Execution Cycle

The *instruction decode function* maps a word-address pair to an instruction as defined in section 2.3 above. The word is the value to be decoded; the address is its location in the memory and is needed to decode instructions that use PC-relative addressing. It is a partial function: some words may have an undefined effect if the microprocessor attempts to execute them. The internal details of the function are of no interest to us here, so we omit the predicate part of the definition.

$$
\boldsymbol{decode} : WORD \times ADDR \rightarrow\!\!\!\!\!+\ INSTRUCTION
$$

The schema $TICK$ describes what happens in one execution cycle. This schema is partial: if the instruction that PC points to has no decoding, the pre-condition of the schema will be false.

$$
\begin{array}{|l}
\hline
\textbf{\textit{TICK}} \\
\quad \Delta PROCESSOR\_STATE \\
\hline
\exists\ instr? : INSTRUCTION \\
\mid \quad (store\ pc,\ pc) \mapsto instr? \in decode \\
\bullet \quad COMPUTE \lor JUMP \lor COND\_JUMP \lor LOAD\_PC \lor STORE\_PC \\
\hline
\end{array}
$$

2.6 Behavioural Model

We now use the schema $TICK$ to define a behavioural model of the microprocessor. This is a description of its input/output behaviour with the details of its internal structure as a state-transition machine abstracted away.

To define the behavioural model we need to define sets to represent the input and output streams. An individual input or output is a function mapping the relevant port addresses to words:

$$\boldsymbol{INPUT} \mathrel{\widehat=} IN\_PORTS \to WORD$$

$$\boldsymbol{OUTPUT} \mathrel{\widehat=} OUT\_PORTS \to WORD$$

An input or output stream is then a series of inputs or outputs indexed by time (measured in CPU execution cycles).

$$\boldsymbol{TIME} \mathrel{\widehat=} \mathbb{N}$$

$$\boldsymbol{IN\_STREAM} \mathrel{\widehat=} TIME \to INPUT$$

$$\boldsymbol{OUT\_STREAM} \mathrel{\widehat=} TIME \to OUTPUT$$

We also need the notion of an execution *history*. This is a time-indexed series of processor states:

$$\boldsymbol{HISTORY} \mathrel{\widehat=} TIME \to PROCESSOR\_STATE$$

The i/o history relation is the ternary relation that relates a triple comprising an input stream, an output stream and an execution history precisely when the following conditions hold: *(i)* the history values may be obtained by successively placing the input values for each time period on the input ports and letting the processor run for one clock tick; *(ii)* the outputs thus obtained at each time period are the ones observed in the history.

$$
\begin{array}{l}
\textbf{\textit{IO\_HISTORY}} \\
\hline
\quad \mathit{inputs} : \mathit{IN\_STREAM}; \\
\quad \mathit{outputs} : \mathit{OUT\_STREAM}; \\
\quad \mathit{history} : \mathit{HISTORY} \\
\hline
\forall\ t : \mathit{TIME};\ \mathit{TICK} \\
\mid \quad \mathit{pc} = (\mathit{history}\ t).\mathit{pc} \\
\wedge \quad \mathit{store} = (\mathit{history}\ t).\mathit{store} \oplus \mathit{inputs}\ t \\
\bullet \quad \mathit{pc}' = (\mathit{history}\ (t{+}\mathit{1})).\mathit{pc} \\
\wedge \quad \mathit{store}' = (\mathit{history}\ (t{+}\mathit{1})).\mathit{store} \\
\wedge \quad \mathit{outputs}\ t = \mathit{OUT\_PORTS} \lhd \mathit{store}'
\end{array}
$$

The behaviour of a processor running a particular program is its input-output relation and belongs to the following set:

$$
\textbf{\textit{BEHAVIOUR}} \mathrel{\widehat=} \mathit{IN\_STREAM} \leftrightarrow \mathit{OUT\_STREAM}
$$

Now we can describe the behavioural model. This is explicitly parametrised by the initial state, i.e. the program to be run.

$$
\begin{array}{l}
\quad \textbf{\textit{behaviour}} : \mathit{PROCESSOR\_STATE} \rightarrow \mathit{BEHAVIOUR} \\
\hline
\forall \quad \mathit{prog}: \mathit{PROCESSOR\_STATE}; \\
\quad \mathit{inputs} : \mathit{IN\_STREAM}; \\
\quad \mathit{outputs} : \mathit{OUT\_STREAM} \\
\bullet \quad \mathit{inputs} \mapsto \mathit{outputs} \in \mathit{behaviour}\ \mathit{prog} \\
\Leftrightarrow (\exists\ \mathit{history} : \mathit{HISTORY} \bullet\ \mathit{history}\ \mathit{0} = \mathit{prog} \wedge \mathit{IO\_HISTORY})
\end{array}
$$

Using the behavioural model, we may now specify rigorously various general properties of programs. As an example, we can now characterise the programs that never run out of control; they are precisely those whose behaviour is a total relation:

$$
\begin{array}{l}
\quad \textbf{\textit{total}} : \mathbb{P}\mathit{PROCESSOR\_STATE} \\
\hline
\quad \mathit{total} = \\
\quad \{\mathit{prog} : \mathit{PROCESSOR\_STATE} \mid \mathit{dom}(\mathit{behaviour}\ \mathit{prog}) = \mathit{IN\_STREAM}\}
\end{array}
$$

2.7 Discussion

The techniques we have used to define the function *behaviour* in this section can readily be adapted to construct a behavioural model from almost any state-based specification. We believe that this approach should be more widely used.

Notions like data and code refinement admit a very direct and clear formulation for a behavioural model. The refinement relation, $\_ \sqsubseteq \_$, can be defined directly in Z as follows:

$$\_ \sqsubseteq \_ : BEHAVIOUR \leftrightarrow BEHAVIOUR$$

$$\forall\; b_1, b_2 : BEHAVIOUR$$
$$\bullet\; b_2 \sqsubseteq b_1 \Leftrightarrow dom\; b_1 \subseteq dom\; b_2 \land dom\; b_1 \lhd b_2 \subseteq b_1$$

That is to say, behaviour b_2 refines behaviour b_1 if, and only if, *(i)* b_2 is defined for all inputs for which b_1 is defined, and *(ii)* every output stream of b_2 on an input admitted by b_1 is also a possible output stream for b_1. These are the usual liveness and safety properties that one requires in refinement.

Refinement rules for the underlying state-based specifications can then be derived in Z as theorems from the above definition rather than posited and justified by metalinguistic methods as is commonly done in the literature.

In the sequel, we will be interested in higher-order properties that have to be expressed with some reference to the internal state. To define these, we will use the i/o history relation. Methodologically, this reflects the following fact: while externally observable behaviour is the ultimate thing of interest, program analysis is carried out using implementation details (the program!); to capture the requirements on a program analysis technique we need some view of those details. We do expect an analysis to have useful consequences at the behavioural level — for example, the basic blocks abstraction that we look at in the next section is expressed in terms of the i/o history relation, but, when it holds, it guarantees that the behaviour relation is total.

3 Basic Blocks Abstraction

3.1 Introduction

The notion of a basic block is well known in the world of compiler design. To quote [1], a basic block is:

> *...a sequence of consecutive statements which may be entered only at the beginning and when entered are executed in sequence without halt or possibility of branch (except at the end of the basic block.*

Compiler designers use decompositions of programs into a set of basic blocks for various kinds of code optimisations. We are concerned with program analysis techniques that deduce a decomposition of a machine code program into a set of basic blocks.

Our intention is that the structure recovered by such a decomposition will enable deeper semantic analyses to be carried out more easily. For example, the REAP reverse engineering tools are able automatically to find program decompositions of a similar sort to the basic block decompositions described here.

These decompositions then justify a translation of machine code into a high level language.

In general, an arbitrary program executing on our microprocessor may admit no useful decomposition into basic blocks; indeed, the program may be self-modifying — making it impossible to distinguish between code and data in the store. However, programs generated by compilers or written by well-disciplined assembly language programmers will normally admit a clear separation of code and data and will have the code structured to admit a decomposition into a set of basic blocks (corresponding to a flow-chart for the program). We are interested in formalising the notion of such a decomposition.

3.2 Representing Basic Blocks

We will need to distinguish between instructions that can cause a transfer of control and the *non-jump instructions* — those for which control just proceeds to the next instructions.

$$\boldsymbol{NON\_JUMP\_INSTRUCTION} \mathrel{\widehat{=}} \mathit{ran\ Compute} \cup \mathit{ran\ StorePC}$$

A *basic block* comprises a (possibly empty) *body* of non-jump instructions followed by an instruction (the *coda* of the basic block) that may cause a transfer of control[3]. The basic block is labelled with the address in memory at which the basic block starts. The following definition captures these aspects of a single basic block. To make formal the full content of the definition given in section 3.1 above, we need to describe a relationship between a whole set of basic blocks and a processor execution history: this is done in sections 3.3 and 3.4 below.

$\boldsymbol{BASIC\_BLOCK}$

$$\begin{array}{l} \mathit{body} : \mathit{seq}\ \mathit{NON\_JUMP\_INSTRUCTION}; \\ \mathit{coda} : \mathit{INSTRUCTION}; \\ \mathit{label} : \mathit{ADDR} \end{array}$$

3.3 Instantaneous Basic Block Decompositions

A basic block decomposition comprises a set of basic blocks which we will require to act as a correct description of the possible control behaviour of a program. To define this requirement, we first define the notion of an *instaneous basic block decomposition*. An instantaneous basic block decomposition is a relation between a set of basic blocks and a processor state. We will develop our specification of this relation in four stages.

[3] It is important that we allow the coda to be a non-jump instruction. E.g., using C as an assembly language, consider the program fragment: "`R1 = 1; L: R1 *= R2; R2 -= 1; if(R2 > 0) goto L;`". A basic block decomposition of this must take the computation instruction "`R1 = 1;`" as the coda of a basic block with an empty body.

1. The basic blocks must correctly describe the instructions encoded in some portion of the memory,

INST_BBD1

$$
\begin{array}{l}
blocks : \mathbb{P}\ BASIC\_BLOCK; \\
PROCESSOR\_STATE \\
\hline
\forall\ b : blocks \\
\bullet\ (\forall i{:}dom\ b.body \bullet \\
\quad (store\ (b.label + i - 1), (b.label + i - 1)) \mapsto (b.body)\ i \in decode) \\
\wedge\ (store(b.label + \#(b.body)), (b.label + \#(b.body))) \mapsto b.coda \in decode
\end{array}
$$

2. No two basic blocks may apply to the same piece of memory:

INST_BBD2

$$
\begin{array}{l}
blocks : \mathbb{P}\ BASIC\_BLOCK; \\
PROCESSOR\_STATE \\
\hline
\forall\ b_1, b_2 : blocks \\
\mid\ \exists i : 0 .. \#(b_1.body); j : 0 .. \#(b_2.body) \bullet b_1.label + i = b_2.label + j \\
\bullet\ b_1 = b_2
\end{array}
$$

3. The program counter must point to some instruction in one of the basic blocks:

INST_BBD3

$$
\begin{array}{l}
blocks : \mathbb{P}\ BASIC\_BLOCK; \\
PROCESSOR\_STATE \\
\hline
\exists\ b : blocks \bullet pc \in b.label .. b.label + \#(b.body)
\end{array}
$$

4. If the processor has reached the end of a basic block, then the next value of the program counter must be the label of one of the basic blocks:

INST_BBD4

$$
\begin{array}{l}
blocks : \mathbb{P}\ BASIC\_BLOCK; \\
PROCESSOR\_STATE \\
\hline
\forall\ PROCESSOR\_STATE'; b : blocks \mid TICK \wedge pc = b.label + \#(b.body) \\
\bullet\ pc' \in \{c : blocks \bullet c.label\}
\end{array}
$$

The conjunction of the above four schemas gives us our definition of an instantaneous basic blocks decomposition.

$$\boldsymbol{INST\_BBD} \mathrel{\widehat=} INST\_BBD1 \land INST\_BBD2 \land INST\_BBD3 \land INST\_BBD4$$

Any state of the microprocessor in which the current and next values of the program counter point to valid instructions admits at least one instantaneous basic block decomposition: namely the degenerate decomposition with just two basic blocks, one for the current instruction and one for the next instruction.

3.4 Correct Basic Block Decompositions

For a basic block decomposition to be useful it must persist over successive execution states. We therefore define a *correct basic block decomposition* for a given initial state (i.e., a given program) to be one which will work as an instantaneous decomposition for ever.

$$\boldsymbol{correct\_bbd} : PROCESSOR\_STATE \to \mathbb{P}\mathbb{P}\ BASIC\_BLOCK$$

$$\begin{array}{l}
\forall\ prog : PROCESSOR_STATE;\ blocks : \mathbb{P}\ BASIC_BLOCK \\
\bullet\ \ blocks \in correct_bbd\ prog \\
\Leftrightarrow (\forall\ inputs : IN_STREAM\ \bullet \\
\quad \exists\ outputs : OUT_STREAM;\ history : HISTORY\ \bullet \\
\quad\quad IO_HISTORY \\
\quad \land\ (history\ 0).pc = prog.pc \\
\quad \land\ (history\ 0).store = prog.store \oplus inputs\ 0 \\
\quad \land\ (\forall\ t : TIME;\ PROCESSOR_STATE \\
\quad\quad \mid\ pc = (history\ t).pc \\
\quad\quad \land\ store = (history\ t).store \oplus inputs\ t \\
\quad\quad \bullet\ INST_BBD))
\end{array}$$

3.5 Discussion

A program possessing a correct basic block decomposition is necessarily total — a fact that is wired into the definition above in a fairly direct way. Moreover, programs that have such decompositions are much nicer than those that don't; in particular, the basic blocks give a clear distinction between code and data in memory; a program with a correct basic block decomposition will neither modify its code nor execute its data[4].

[4] This "Harvard" property could be expressed directly in terms of the behavioural model without introducing the basic blocks. We have introduced the basic blocks precisely because the program analysis techniques of interest produce decompositions of this sort as part of their output and we are concerned with formalising exactly what that output means.

The notion of a correct basic block decomposition as specified above is directly applicable to embedded systems that run a fixed program held in ROM. It would also apply with a little elaboration to a system that loads a fixed program into RAM at boot time — the above definition would need to be modified to allow the basic block decomposition not to take effect until the execution cycles that load the program are complete.

A multiprocessing operating system cannot be expected to satisfy the above definition directly; however, the virtual machine provided by an operating system to each process could well permit a correct basic block decomposition. For example, of the hundreds of programs that can be run on the Linux system that I am using to prepare this paper, only a handful of specialist programs (like interactive compilers) would be expected to modify their own code.

4 Concluding Remarks

4.1 Limitations of the Processor Model

A number of important features of microprocessors in the real world have not been addressed by the specification in this paper. To list a few:

Multi-word instructions: to handle these simply requires the decode function to take as its argument a sequence of memory words and to return the number of memory words actually occupied by the decoded instruction for use in adjusting the program counter while executing the non-jump instructions.

Multiple word lengths: most real microprocessors support several different word lengths (e.g., 8-bit, 16-bit and 32-bit words for the Intel x86 and the Motorola 680x0). Multiple word lengths can be dealt with quite simply in a variety of ways (see chapter 9 of [3] for one approach).

Multiple reads & writes: instructions with a built-in loop, such as the Z80's block transfer instructions or the x86's repeated string-to-string-copy instructions, could be handled by modifying the schema $TICK$ to execute the loop.

Pipelining: In some RISC processors and microcode engines, side-effects of the decode/execute pipeline are visible to the programmer. In the SPARC 9 architecture for example, while a jump instruction of a certain type is being executed, the instruction immediately following the jump is being decoded and will be executed on the next cycle[5]. The pipeline would have to be included explicitly in the execution model to cover such an architecture.

Interrupts: in a sense, handling interrupts amounts just to including the program counter (and any other registers affected) as a possible input port. However, to have abstractions like the basic block decomposition work correctly, it would probably be better to model interrupts as an additional kind of instruction and to include the microprocessors interrupt lines in the model.

[5] So, for example, subroutine exit on this architecture is implemented by a return-from-subroutine instruction immediately followed by the instruction that restores register values for the calling routine.

With the possible exception of interrupts, none of these features should prove a major source of additional complexity. Moreover the extra complexity is mostly in the traditional state-based part of the model, so that one may readily exploit techniques that have been proposed in the literature (see [3] for a literature guide). The SPARC 9 model which we outline in section 4.2 below addresses both multiple word lengths and pipelining.

4.2 Recent Work

Interglossa have recently undertaken a research project sponsored by DERA, Malvern to work on Chris Sennett's Template Analysis [4] — a technique for analysing the use of pointers in C code. The result of the Template Analysis is, in effect, a type assignment for a C program in a type system that is "tighter" than that imposed by the C language rules.

Part of Interglossa's research was to push the type assignment resulting from the Template Analysis through to the assembly language level, giving a way of interpreting low-level entities in higher level terms. Based on the specification in the present paper, Tom Lake developed a formalisation of the Sun Microsystems SPARC 9 architecture to guide the research.

The specification of the SPARC 9 model together with "validity conditions" and a description of the template typing for assembly code took about 30 pages of Z. The validity conditions are similar in general spirit to the basic blocks abstraction discussed above but extended to cover separation of data and code, orderly control flow, and correct use of the stack and of global data.

In the technical document of 1997 in which the specification in this paper was first written up, I wrote:

> *The specification could be used in two ways to model a real microprocessor:* (i) *its ideas could be re-used to model the specifics of the microprocessor;* (ii) *it could be used as a sort of microcode engine to describe the semantics of the microprocessor.*

On reading the Interglossa specification two years later, it was interesting to see how these ideas had stood the test of time. For the SPARC 9 example, it turned out to be easiest to do a mixture of *(i)* and *(ii)*. The main framework for the store model re-used the general ideas of section 2.1 above but recast to cover the specific details of the SPARC 9 memory model. However, to simplify the execution model, individual SPARC 9 instructions were handled as sequences of "microcode instructions" very similar to those described in section 2.3.

4.3 Future Research

Lemma 1 and Interglossa hope to collaborate soon to continue this line of research. There have been considerable advances in program analysis theory and practice in recent years [7]. We are particularly interested in investigating how program analysis techniques and formal specification and verification technology can interact to make automated or semi-automated validation of compiled code a viable method.

Acknowledgments

I am indebted to Tom Lake of Interglossa and to Mike Hill of the Defence and Evaluation Research Agency, Malvern, for their assistance in the preparation of this paper. The referees' comments were most helpful and have resulted in improvements both to the specification and to its presentation.

Index of Z Global Variables

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.

[2] R.D. Arthan. Mechanizing Proof for the Z Toolkit. To appear; available on the World Wide Web at `http://www.lemma-one.com/papers/papers.html`, presented at the Oxford Workshop on Automated Formal Methods, June 1996.

[3] Jonathan Bowen. *Formal Specification and Documentation using Z: a Case Study Approach.* International Thomson Computer Press, 1996.

[4] M.G. Hill and C.T. Sennett. Final Report of Integrity Methods for Commercial Software, DERA Technical Report DERA/CIS/CIS3/TR980138/1.0. Technical report, Defence Evaluation and Research Agency, Malvern, 1998.

[5] D.J. King and R.D. Arthan. Development of Practical Verification Tools. *Ingenuity — the ICL Technical Journal*, 1996.

[6] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[7] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 1998.

[8] J.M. Spivey. *The Z Notation: A Reference Manual, Second Edition*. Prentice-Hall, 1992.

[9] Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement, and Proof*. Prentice/Hall International, 1996.