A Verified Compiler for a Structured Assembly Language

Paul Curzon

University of Cambridge
Computer Laboratory
New Museums Site
Pembroke Street
Cambridge
CB2 3QG
United Kingdom

In Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications,
M. Archer, J. Joyce, K. Levitt and P. Windley (Editors)
IEEE Computer Society Press, 1992

A Verified Compiler for a Structured Assembly Language

Paul Curzon
University of Cambridge,
Computer Laboratory

Abstract

We describe the verification of a compiler for a subset of the Vista *language: a structured assembly language for the* Viper *microprocessor. This proof has been mechanically checked using the* HOL *system. We consider how the compiler correctness theorem could be used to deduce safety and liveness properties of compiled code from theorems stating that these properties hold of the source code. We also show how secure compilation can be achieved using automated theorem proving techniques.*

1 Introduction

In this paper, we describe the verification of a compiler for a subset of the Vista language[10]. Our motivation for verifying the compiler is to allow us to infer properties about the code which is actually executed from properties we prove about Vista programs. Previous work on the formal verification of compilers has largely considered the compiler correctness theorem itself to be the ultimate goal. Consequently, little attention has been given to identifying the way in which the correctness theorem will be used. We describe how our compiler correctness theorem could be used to infer the safety and liveness of code produced by our compiler from the safety and liveness of the source code. We also consider how the compiler correctness theorem required is affected by the properties we wish to infer. All the proofs described here have been mechanically checked by the HOL system[5]. We assume some familiarity with HOL and higher-order logic, though a knowledge of first-order logic should be sufficient to follow the main points of this paper.

Vista is a commercially available structured assembly language designed for use with the Viper microprocessor[4] at the Defence Reseach Agency (Electronics division), formerly RSRE. Viper is a 32-bit computer, aspects of which have been formally verified[2, 3]. The statements of Vista are Viper machine instructions together with structural commands such as while loops. The ultimate aim of the project is to build a verified implementation of the Vista language for the Viper microprocessor. In the work completed so far, the target language for the compiler we have considered is a simple flat assembly language, Vip, for a Viper-like microprocessor. Vip is idealized in that it assumes infinite word and memory sizes. We have also only considered a subset of the Vista language.

The semantics of the source and target languages are defined in higher-order logic, as is the compiler. The HOL system does not currently provide tools for executing general higher-order logic definitions. Despite this we illustrate that the compiler can be "executed" securely using theorem proving techniques.

This work builds upon the work of Joyce to verify the correctness of a compiler for the Tamarack microprocessor[8]. The source and target languages considered by Joyce are different to those used here. However, the structure of the proofs are similar and both have been mechanically checked using the HOL system. Joyce was largely concerned with linking the compiler proof with a proof of correctness of the underlying machine[7], rather than with proofs of higher level programs as here. This difference in emphasis prompted us to use a slightly different correctness statement.

Stepney *et al.* have also considered the correctness of a compiler for the Viper microprocessor[13]. They were interested in demonstrating that the compiler was correct by construction. The source language was of a similar complexity to the subset of Vista considered here, though in addition supported procedure calls and a simple model of I/O. However, their proof work was informal and not machine checked. The

work was completed in a matter of days, but as they remark *"... a properly rigorous development of a compiler for a full language would take longer!"*[13].

Substantial work on the compiler correctness problem has been performed at Computational Logic Inc. A code generator from Micro-Gypsy to Piton has been verified using the Boyer-Moore system[14]. An assembler and linking loader for Piton has also been verified[12]. These components are part of a system stack verified down to the hardware level. The Gypsy Verification Environment which implements the Floyd-Hoare approach to verification is also available. However, using it has a serious disadvantage. There is no assurance that the semantics of Micro-Gypsy used by the Verification Environment are the same as those used in the compiler proof. As an alternative it is suggested that proofs be performed using the operational semantics used in the compiler verification. This unfortunately loses the advantages of the Floyd-Hoare approach and the infra-structure provided by the Gypsy Verification Environment. One of the issues addressed in this paper is the linking of a compiler proof to proofs of programs using programming logics.

Much other work has appeared concerning the compiler correctness problem. A good survey of this work has been given by Joyce[8].

2 The source and target languages

2.1 Vista: its syntax and semantics

The source language we have considered is a subset of Vista. Since Vista is an assembly language, the general purpose registers are accesible and the basic commands correspond directly to those available on the underlying machine; here Vip. Vista also has properties of a high level language. Variables may be declared and used, and structured commands such as while loops are provided.

Figure 1 gives the syntax of the programs in the subset of Vista considered. A program consists of zero or more variable declarations followed by a command. Commands supported are skip, stop, register and variable assignments, ALU operations, sequencing and while loops. The conditions which may be used in the test part of the while loop are comparisons between a register and an expression, and direct tests of the B register: a single bit register which holds condition codes. Expressions may be named variables or natural numbers. Three general purpose registers are accesible: the A, X and Y registers.

The syntax is defined using the type definition package provided with the HOL system[11]. The parser support of the HOL system is used to parse the concrete syntax shown in the figure into the abstract syntax as defined by the type definition package. For example, the concrete syntax

```
A := 2
```

is parsed by the HOL system to the underlying abstract syntax

```
ASSIGN A (NUM 2)
```

Throughout this paper I will use the concrete syntax for convenience.

As only an informal semantics of Vista existed, part of this work has involved the production of the formal semantics. Though only a small subset has been considered, the semantics contains some interesting features. Conditions are problematic since they side-effect a value into the B register. The actual value placed in the register is not defined in the Vista language, but is implementation dependent. The stop command also causes complications. It can be used to terminate the program at any point. The semantics must handle such termination even at inner levels of loops.

A denotational semantics was used. The semantics of a program p is given as a relation, $\mathcal{P}[\![\mathtt{p}]\!]$, between states. The relation is true if the program terminates in the final state from the given initial state.

A Vista state may be one of three kinds. An *Error* state indicates that the program was invalid. This could, for example, be due to variables being used without being declared. A *Halt* state indicates that the program has terminated. This is used to prevent further processing being performed once a stop command has been executed. All commands leave a halt state unmodified. This is how termination before the textual end of the program is handled. A *Run* state indicates that the program is executing normally. It is an intermediate processing state. The latter two kinds of state consist of a memory store and register store, giving the values of memory and registers, respectively. States are defined using the type definition package:

```
VistaState = ERROR | HALT ms rs | RUN ms rs
```

Throughout this paper we use the variables q, q1 and q2 to range over Vista states.

The purpose of the declarations in a program is to build an environment. An environment relates names in the program to the locations in memory store that they will represent. The declarations are evaluated to give an environment which is passed to the command.

```
<P> ::=                                  Programs
        PROGRAM name <D>; <C> FINISH |
        PROGRAM name <C> FINISH

<D> ::=                                  Declarations
        DATA <V> |                           variable declaration
        <D> ; <D>

<C> ::=                                  Commands
        SKIP |
        STOP |
        <R> := <E> |                         register assignment
        <V> := <R> |                         variable assignment
        <R> := <R> op <E> |                  ALU operations
        <C> ; <C> |
        WHILE <B> DO <C> OD

<B> ::=                                  Conditions
        B |                                  test B register
        <R> comp <E>                         comparisons

<R> ::= A | X | Y                        Registers

<E> ::= <V> | num                        Expressions

<V> ::= string                           Variables
```

Figure 1: The Syntax of the Vista subset

The semantics of a declaration is therefore a function which maps an environment to an environment. The initial environment is defined to be empty. As each variable declaration is encountered, a new entry is added.

The denotation of a command is given by a relation taking an environment, an initial state and a final state. It is true if executing the command in the environment from the initial state results in the final state. To illustrate the semantics of commands, we will look at three simple commands: Skip, Stop and Sequencing.

Consider the denotation of the Skip command. It has no effect on the state, whatever the environment, so the initial and final states should be identical.

```
C⟦SKIP⟧ env q1 q2 ≝ (q2 = q1)
```

The STOP command is slightly more complex.

```
C⟦STOP⟧ env q1 q2 ≝
   (q2 =
     (IsRun q1 ⇒
       HALT (RsOf q1) (MsOf q1) |
       q1))
```

If the initial state is a Run state, the register store and memory store are extracted and a Halt state formed. If the initial state is not a Run state, the STOP instruction does not alter it. Again the environment is not used since no memory accesses are made by the command.

If two commands are sequenced, then the first should be executed followed by the second.

```
C⟦c1; c2⟧ env ≝
   SemSeq (C⟦c1⟧env) (C⟦c2⟧env)
```

where

```
SemSeq sem1 sem2 q1 q2 ≝
    ∃q.  (sem1 q1 q) ∧ (sem2 q q2)
```

The first command must return some intermediate state, q, from which the second command returns the final state. Both commands use the same environment as it is provided externally and not altered by the commands themselves.

Conditions are problematic since they both return the result of the test and place a new value in the B register as a side-effect. This is handled by splitting the denotation of each condition into two parts. The result of the test is given by a function on the initial state. The permissible state changes are then described by a relation as for the semantics of commands. For example, the semantics of the B condition is described by a relation SemB between an initial and final state, and a function SemBansB which maps an initial state to the value of the test.

Matters are complicated further by the fact that the value placed in the B register is not defined by the semantics of Vista. In any given situation, the value may be different for different implementations. Despite this, the B register is accesible to the Vista programmer. In particular, it can be directly tested using the B condition. To deal with this an explicit undefined value is introduced which is placed in the B register during the test.

If the B register is tested after an undefined value has been side-effected into it, different implementations of Vista could produce different results. Since Vista was designed for use in safety-critical applications, we decided when defining the formal semantics that it should be a program error to test the B register after an undefined value had been placed there. This is reflected in the semantics by returning an error state. The Vista programmer is given the proof obligation of ensuring that this situation cannot arise in a given program. This is more restrictive than absolutely necessary. A program could test the B register without its later behaviour being dependent on the test. All implementations would therefore exhibit the same behaviour on such a program. However, it is considered that such an anomaly should be detected and removed from a program intended for safety-critical applications.

2.2 Vip: its syntax and semantics

The target language for the compiler, Vip, is a flat assembly language for a Viper-like microprocessor. We compile to this intermediate assembly language rather than directly to machine code for several reasons. It allows us to split the proof requirements into manageable and independent tasks. For example, in the first pass of the compiler we do not consider finite word and memory size. It also means the proofs are reusable. We can use the proofs of the first pass for different underlying machines. Similarly, if a proof of correctness of the second pass of the compiler were performed it could be reused for other source languages. The above reasons have been previously identified by Joyce[8]. Furthermore, an optimising pass could be added to the compiler at the flat assembly language level, and a proof of its correctness combined with the proofs for all source and target languages.

Seven assembler instructions are supported: Stop, Jump, Conditional Jump, Load, Store, a generic ALU

```
<I> ::=                              Instructions
        STP |                            Stop the Processor
        JMP <L> |                        Jump to given address
        JNB <L> |                        Jump to given address if the B register is false
        LOAD <R> <O> |                   Load a register with the Operand value
        STORE <R> <W> |                  Store the register value at the given address
        ALU op <R> <R> <O> |             Do an ALU operation between a source register and
                                         operand, storing the result in a destination register
        COMP comp <R> <O>                Compare a source register and operand, storing the
                                         result in the B register

<O> ::=                              Operands
        LITERAL num |                    A literal value
        CONTENTS num                     The address of the value

<W> ::= AT num                       Write Addresses

<R> ::= num                          Registers

<L> ::= num                          Labels
```

Figure 2: The Syntax of Vip

instruction and a generic comparison instruction. The syntax of the language and its informal semantics are given in Figure 2.

The formal semantics of Vip is given as an operational semantics in higher-order logic. A Vip state is represented as a tuple holding the values of the memory and registers, `(ram,p,a,x,y,b,stop)`. The elements of this tuple correspond to the memory, program counter, accumulator, X register, Y register, condition flag and stop register. The program is stored in a separate read-only code store, which does not form part of the state. Throughout this paper we use variables `v`, `v1` and `v2` to range over Vip states.

The semantics of a Vip program is given by a relation `Vip`.

```
Vip rep cs end v1 v2 ≝
   ∃n.
    (v2 = Vipn n rep cs v1) ∧
    (CodeExecutesInNsteps rep n cs v1 end)
```

It takes as arguments a representation function (which provides the semantics of generic operators: this is discussed in more detail in Section 2.3), a code store, the address which marks the end of the program, and initial and final states. The relation will be true if the program can be executed from the initial state in some finite number of cycles to reach the final state. In addition, the program should either halt the processor or jump beyond the end address for the first time on the last cycle. This condition is expressed by the relation `CodeExecutesInNsteps`. It is assumed that the program resides in contiguous addresses in the code store.

The relation `Vipn` returns the final state after executing a program for a given number of cycles from some initial state. A single cycle of the program consists of checking that the processor has stopped and, if not, fetching the instruction currently addressed by the program counter, then decoding and executing it. The semantics of each instruction is given by a state transition function. For example that for the Jump instruction is given below.

```
VipSemJmp label (ram,p,a,x,y,b,stop) ≝
   (ram,label,a,x,y,b,stop)
```

It updates the program counter with the label given as an operand.

2.3 Language schemas

In the previous discussion we glossed over the precise details of the ALU and comparison operations provided by the source and target languages. We have in effect defined source and target language schema, so that these details do not need to be addressed. Using the methods suggested by Joyce[9], the definitions are completely generic with respect to the ALU operations and comparisons. For example, a single generic command is given to cover all Vista

machine functions. Its syntax takes an ALU operation argument, *op*.

R := R *op* E

The syntax of the corresponding Vip command has a similar ALU operation argument.

ALU *op* R R O

In both cases the type of *op* is given by a type variable `*aluop`. Particular source and target languages are obtained by instantiating this type variable with a suitable type. For example, if we instantiated `*aluop` to the type `Alu` given below we would have a source and target language with Addition, And and Or operations.

```
Alu = ADD | AND | OR
```

The semantics of these operations must be provided separately by a representation function. It is a function from type `*aluop` to type (`num->num->num`)—the type of ALU operations. In the above example, it might map `ADD` to integer addition functions and `AND` and `OR` to the appropriate bitwise functions.

This allows a single Vista semantic predicate and a single Vip semantic function to be defined to cover all ALU operations. They extract the semantics of particular operations from the representation function which is passed to them as an argument. This means that we only need perform proofs once for the ALU command rather than once for each available operation. By giving the ALU command a variable type, we can reuse the proof for a range of source and target languages within the Vista/Vip family. This reflects the fact that the correctness of the compiler does not depend on the particular source operations available. They are compiled directly to target ones with identical semantics. The above discussion also applies to the comparison operators which are treated in a similar way.

3 The compiler

The compiler is defined in higher-order logic by a function `CompileProgram` which translates Vista programs to Vip assembly code. It returns one of two kinds of value: either the compiled code or a compile-time error. Compiled code is represented as a list of generic instructions.

```
Code = COMPILE_ERROR |
       CODE (((*compop,*aluop)Inst) list
```

A compile-time error can occur, for example, if undeclared variables are used.

The compiler first creates a symbol table from the declarations. This is similar to the environment constructed by the semantics. It maps variable names to locations in Vip memory where the variable's value will be stored. The symbol table is passed to the command translator `TransCom`.

The simple non-structural commands such as stop and machine functions translate to single Vip commands; expressions and registers having been first translated. For example, the translation of the machine functions is given by the following definition.

```
TransCom
   (dest := src OP exp)) symbtab base ≝
     TransMchFunc
      OP
      (TransReg src)
      (TransReg dest)
      (TransExp exp symbtab)
```

This is a *generic* translation function. It is used to translate all machine functions. It extracts the operator from the source command and passes it along with the translation of the registers and expression to the function `TransMchFunc` defined below. Registers are translated to the corresponding number. Expressions are passed the symbol table, so that the memory locations corresponding to variables can be looked up. The argument `base` is not used for the translation of this command. It gives the address to which the code will ultimately be loaded, and is used by structural commands such as the while loop whose translation includes jump instructions, which require actual addresses.

```
TransMchFunc OP s d e ≝
   ((e = ERROR) =>
     COMPILE_ERROR |
     LoadInst (ALU OP d s (ErrValOf e)))
```

If the translation of the expression is an error a compile-time error results. Otherwise, the result is extracted using `ErrValOf`, and a Vip `ALU` command formed. This is turned into compiled code using the function `LoadInst`.

When translating structural commands subcommands are first translated and the results combined. For example, the translation of the while loop is given by the following definition.

```
TransCom
   (WHILE b DO c OD)) symbtab base =def
     TransWhile
      (TransBcom b symbtab)
      (TransCom c symbtab
        (base + (SizeOfB b)))
      base
      (SizeOf (WHILE b DO c OD))
```

The condition and body are first translated. The base address of the command will be the base address of the while loop plus the size of the code for the condition. The function `TransWhile` is defined below. It is also passed the size of the translated code so that it can calculate the loop exit. The size is computed recursively, using the function `SizeOf`.

```
TransWhile bcode ccode base size =def
   (Combine bcode
   (Combine (LoadInst (JNB (base + size)))
   (Combine ccode
            (LoadInst (JMP base)))))
```

The compiled code of the while loop consists of the code for the condition followed by a conditional jump out of the loop. This is followed by the command for the body, and a jump back to the start of the loop.

3.1 Compilation by theorem proving

The purpose of verifying the compiler is to increase our confidence that the compiled code ultimately executed has the same semantics as the source code. This will not be achieved unless we can use the verified compiler to perform the compilation. No tools are provided by the HOL system for executing higher-order logic definitions (though some work has been performed on executing subsets of the logic using ML[1]). It might therefore seem that our verified compiler is of no use. The same effect as executing the compiler can however be achieved using theorem proving techniques. Given a Vista program p and a base address, *base*, to which the code is to be compiled, we have derived a rule (a conversion) which returns a theorem of the form

$\vdash$ `CompileProgram` p *base* `=` *compiled code*

where *compiled code* is the compiled version of program p and is generated as part of the theorem proving process. An example of such a theorem is given in Figure 3.
The conversion performs this proof completely automatically. This is done by taking the compiler definition, specialising it with the source program and base address values, and rewriting with the definitions of the compiler.

We thus prove a theorem that the generated compiled code is the result of performing the compilation as given by the definition of the compiler. By verifying the compiler, we also prove that this definition preserves the meaning of the program. The compiled code given in the theorem can therefore be considered secure in the sense that it will have the same semantics as the source program.

Unfortunately, this method of compilation is very slow. It would only be practical to use it when producing the final production version of the code. This would ensure that the final code had the same semantics as the source code. A faster, though insecure compiler could be used during program development. All tests should be re-executed using the production version of the code to ensure that any increased assurance in the correctness of the code gained by carrying out the tests during development is not lost.

4 Safety and liveness

To determine a suitable form for the compiler correctness statement we must examine the way it will be used. In particular we need to consider the correctness properties we would like to prove about source programs and consequently infer about the code produced by the compiler. The properties normally associated with program correctness are *safety* and *liveness*. Safety is partial correctness. If a state satisfies some precondition, and is taken by the semantics to some final state, then a given postcondition will hold of the final state. This will always be true for programs which do not terminate (i.e., in which there is no final state which satisfies the semantics).

```
Safe P semantics Q =def
   ∀q1 q2.
    P q1 ∧ semantics q1 q2 ⊃ Q q2
```

Liveness is total correctness. If the precondition holds, there will be a final state satisfying both the postcondition and semantics (i.e., the program will terminate and in a state satisfying the postcondition).

```
Live P semantics Q =def
   ∀q1.
    P q1 ⊃ ∃q2.  (semantics q1 q2 ∧ Q q2)
```

For some semantics, safety can follow from liveness. This is so when for each initial state there is a *unique* final state satisfying the semantics. A semantics of this form is termed *deterministic*.

```
⊢ CompileProgram(
    PROGRAM wombat
     A := 0; X := 0;
     WHILE X < 5 DO
      A := A + 2; X := X + 1
     OD;
     STOP
    FINISH) 0 =
     CODE
      [LOAD 0 (LITERAL 0);
       LOAD 1 (LITERAL 0);
       COMP < 1 (LITERAL 5);
       JNB 7;
       ALU + 0 0 (LITERAL 2);
       ALU + 1 1 (LITERAL 1);
       JMP 2;
       STOP]
```

Figure 3: An Example of a Compilation Theorem

```
Deterministic semantics ≝
   ∀q1 q2 q3.
    (semantics q1 q2) ∧
    (semantics q1 q3) ⊃
     (q2 = q3)
```

The semantics of the Vista subset given is deterministic. To prove the correctness of a Vista program we therefore need only prove that it is live.

It is ultimately the Vip code produced by the compiler which will be executed. The purpose of the compiler correctness theorem is to allow us to deduce the correctness of the compiled code from the correctness of the Vista program. We would like to use the correctness statement to derive an inference rule which makes this deduction automatically. Informally, it should have the following form, where `vista` is the semantics of the vista program, and `vip` the semantics of the compiled version:

$$\frac{\vdash \texttt{Correct vista}}{\vdash \texttt{Correct vip}}$$

Since the semantics of Vip is also deterministic, the notion of correctness we need in the above is liveness. Safety for the compiled program will then follow.

For a Vista program, the definition of liveness is as given above. For the compiled code it is more complicated, since we wish to use the same precondition and postcondition as for the Vista program. These will be defined in terms of Vista states rather than Vip ones. If the relation $\approx$ relates non-error Vip states to the corresponding Vista state then the liveness definition will need the form shown below.

```
VipLive P vip Q ≝
   ∀v1 q1.
     P q1 ∧ (v1 ≈ q1) ⊃
      ∃v2 q2.  vip v1 v2 ∧
               (v2 ≈ q2) ∧
               Q q2
```

In reality the definition is more complex, since it must take account of error states and the loading of compiled code. Rather than pass the semantics of the code we pass a triple consisting of the code, base address and code size.

```
VipLive P (code,base,size) Q ≝
   ∀v1 q1.
    P q1 ∧
    ((v1 ≈ q1) ∨ (q1 = ERROR)) ∧
     (CodeLoaded
       base (base + size) code cs) ∧
     (Ploaded rep v1 base) ⊃
      ∃v2 q2.
       Q q2 ∧
       ((Vip rep cs (base + size) v1 v2 ∧
        (v2 ≈ q2)) ∨ (q2 = ERROR))
```

The code produced by the compiler `code` must be loaded to a suitable location in the code store of Vip. The program counter must also initially hold this location. These conditions are given by the predicates `CodeLoaded` and `Ploaded`, respectively.

Once we have the correctness inference rule described above, from theorems giving the correctness of Vista programs we can infer equivalent results about

the compiled code. Suppose a liveness theorem of the form

$$\vdash \texttt{VistaLive}\ P\ (\mathcal{P}[\![p]\!])\ \mathrm{Q}$$

had been proven, where P and Q are the precondition and postcondition, respectively, of program p with semantics $\mathcal{P}[\![p]\!]$, and we had also proved that the program compiles successfully. We could use the derived inference rules to deduce the liveness of the compiled code.

```
⊢ VipLive
    P
    (CompileProgram p base, base, SizeOf p)
    Q
```

Using the techniques suggested by Gordon[6] future work could involve deriving a programming logic from the semantics of Vista based on the definition of liveness. This would enable the proofs of correctness of Vista programs to be performed using Hoare style proof. Since the programming logic would use the same semantics as that for the compiler correctness proof, there would be no loss of security.

5 The compiler correctness statement

Several different forms of compiler correctness theorem could be proved. If we were only interested in the safety of programs, a suitable correctness theorem would have the form:

```
compiled code semantics ⊃
     source program semantics
```

This is not sufficient when considering liveness properties, which requires the form:

```
source program semantics ⊃
     compiled code semantics
```

If we are interested in both safety and liveness we would prove an equality:

```
compiled code semantics =
  source program semantics
```

This turns out to be stronger than is required for deterministic target languages, however. As discussed above, safety then follows from liveness. We therefore need only prove a correctness theorem of the second form to obtain both the results we desire.

A more detailed version of the correctness theorem we use is given below:

```
∀q1 q2 v1.
   vista q1 q2 ∧ (v1 ≈ q1) ⊃
     ∃v2.  vip v1 v2 ∧ (v2 ≈ q2)
```

The semantics of the Vista program is given by `vista` and that of the compiled code by `vip`. The relation $\approx$ once more relates Vip states to the corresponding Vista state. This correctness theorem states that if the semantics of a Vista program is such that if an initial state `q1` results in a final state `q2`, where `q1` corresponds to a Vip state `v1`, and the compiled code is executed for a sufficient number of cycles, a Vip state `v2` will result which corresponds to `q2`.

For the same reasons as with the definition of liveness, the actual compiler correctness statement used, shown below is more complicated still.

```
CompilerCorrectness code base size ≝
   ∀q1 q2 v1 cs.
    (CodeLoaded base (base + size) code cs) ∧
    (Ploaded rep v1 base) ∧
     vista q1 q2 ∧
     (v1 ≈ q1) ∨ (q1 = ERROR) ⊃
      (q2 = ERROR) ∨
      ∃v2.  Vip rep cs (base + size) v1 v2 ∧
            (v2 ≈ q2)
```

A formal proof of a correctness theorem of this form has been constructed using the HOL system for the code produced by our compiler.

```
⊢ ∀p base.
   CompilerCorrectness
     (CompileProgram p base)
     base
     (SizeOf p)
```

The proof proceeds by structural induction over the Vista commands.

6 Conclusions and further work

In summary, we have used the HOL theorem proving system to prove a generic compiler correctness theorem for a compiler for a subset of the Vista language. We have illustrated the usefulness of this theorem by deriving an inference rule which deduces safety and liveness properties of compiled code from properties of the source code. We have also shown how theorem proving may be used to perform compilation. This helps ensure that the compiled code used is that for which the correctness results were obtained.

The target language for the compiler was a simplified assembly language. Work is in progress to redo

the proof for a realistic assembly language, Visa, for the Viper microprocessor. This proof will be for a larger subset of Vista which includes input and output commands, and will also use a generic word type, rather than the number type used in the proof presented. A proof of correctness of a translator from Visa to Viper machine code will then be performed. We also intend to define a programming logic for Vista as described. This would allow proofs of correctness of Viper machine code to be inferred from Hoare style proofs of correctness of Vista programs.

Acknowledgements

Mike Gordon helped me to understand the differing needs of safety and liveness in addition to giving me much general advice and support. I am grateful to Jeff Joyce and Clive Pygott for their help and encouragement. John Kershaw helped iron out the details of the semantics of Vista. Tom Melham prompted me to think more closely about the form the compiler correctness statement should take. Gavin Bierman made many useful comments about an early draft of this paper. I would also like to thank everyone in the Hardware Verification Group at Cambridge, for the many stimulating meetings and discussions.

This work has been funded by MoD research agreement AT2029/205.

References

[1] Albert John Camilleri. Executing behavioural definitions in higher order logic. Technical Report 140, PhD Thesis, University of Cambridge, Computer Laboratory, February 1988.

[2] Avra Cohn. A proof of correctness of the Viper microprocessor: The first level. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 1–91. Kluwer Academic Publishers, 1988.

[3] Avra Cohn. Correctness properties of the Viper block model: The second level. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 27–72. Springer-Verlag, 1989.

[4] W. J. Cullyer. Implementing safety critical systems: The Viper Microprocessor. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 1–25. Kluwer Academic Publishers, 1988.

[5] Michael J. C. Gordon. HOL: A proof generating system for higher order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128. Kluwer Academic Publishers, 1988.

[6] Michael J. C. Gordon. Mechanizing programming logics in higher order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 387–439. Springer-Verlag, 1989.

[7] Jeffrey J. Joyce. Totally verified systems: Linking verified software to verified hardware. In M. Leeser and G. Brown, editors, *Specification, Verification and Synthesis: Mathematical Aspects*. Springer-Verlag, 1989.

[8] Jeffrey J. Joyce. A verified compiler for a verified microprocessor. Technical Report 167, University of Cambridge, Computer Laboratory, March 1989.

[9] Jeffrey J. Joyce. Generic specification of digital hardware. Technical Report 90–27, The University of British Columbia, Department of Computer Science, September 1990.

[10] J. Kershaw. Vista user's guide. Technical Report 401–86, The Royal Signals and Radar Establishment, 1986.

[11] Thomas F. Melham. Automating recursive type definitions in higher order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 341–386. Springer-Verlag, 1989.

[12] J. Strother Moore. A mechanically verified language implementation. *Journal of Automated Reasoning*, 5:461–492, 1989.

[13] Susan Stepney, Dave Whitley, David Cooper, and Colin Grant. A demonstrably correct compiler. *Formal Aspects of Computing*, 3:58–101, 1991.

[14] William D. Young. A mechanically verified code generator. *Journal of Automated Reasoning*, 5:493–519, 1989.