Coq: The world’s best macro assembler

Andrew Kennedy\(^1\), Jonas Jensen\(^2\), and Nick Benton\(^1\)

\(^1\) Microsoft Research Cambridge
\(^2\) ITU Copenhagen

Abstract. We describe a Coq formalization of a subset of the x86 architecture. One emphasis of the model is brevity: using dependent types, type classes and notation we give the x86 semantics a makeover that counters its reputation for baroqueness. We model bits, bytes, and memory concretely using functions that can be computed inside Coq itself; concrete representations are mapped across to mathematical objects in the ssreflect library (naturals, and integers modulo \(2^n\)) to prove theorems. Finally, we use notation to support conventional assembly code syntax inside Coq, including lexically-scoped labels. Ordinary Coq definitions serve as a powerful “macro” feature for everything from simple conditionals and loops to stack-allocated local variables and procedures with parameters. Assembly code can be assembled within Coq, producing a sequence of hex bytes. The assembler enjoys a correctness theorem relating machine code in memory to a separation-logic formula suitable for program verification.

1 Introduction

The Coq proof assistant \([14]\) has proved remarkably versatile, with applications that span deep, research-level mathematics \([6]\), programming language metatheory \([1]\) and compiler correctness for realistic languages \([8]\). As part of a larger project tackling verification of systems software, we have used Coq to model a subset of the x86 machine architecture, generate binary code for it, and specify and prove properties of that code \([7]\). This paper concerns the first two of these tasks, showcasing Coq as a rich language for giving very readable (and executable) semantics for instruction set architectures, and as a means of writing assembly language programs and furthermore generating machine code from those programs. Using the semantics and assembler as a foundation, the full power of Coq’s proof capabilities can be brought to bear on low-level programs.

1.1 An example

Figure 1.1 presents code that computes the factorials of 10 and 12 and prints them to the console using a function that is provided as its only argument. (So in C notation, its signature is `void main(void (*printnum)(int)).` When we run `coqc` over this file, we get a hexadecimal listing.
Definition call_cdecl f arg :=
  push arg;; CALL f;; add ESP, 4.

Definition main :=
  let_simple fact arg :=
    mov EAX, 1;; mov ECX, 1;;
    while (cmp ECX, arg) CC_LE ( (* while ECX <= arg *)
      mul ECX;; (* Multiply EAX by ECX *)
      inc ECX)
  in
    push EBP;; mov EBP, ESP;;
    call_simple fact 10;; call_cdecl [EBP+8] EAX;;
    call_simple fact 12;; call_cdecl [EBP+8] EAX;;
    pop EBP;; ret 4.

Definition bytes := assemble_total #x"C0000000" main.

Goal True.
  let b := eval compute in (bytesToHex bytes) in idtac b. done. Qed.

>coqc fact.v
"E9 1D 00 00 00 B8 01 00 00 00 09 01 00 00 00 E9 04 00 00 00 F7 E1
FF C1 3B CA 0F 8E F4 FF FF FF E7 55 8B EC BF 34 00 00 C0 BA 0A
00 00 00 E9 D1 FF FF FF 55 08 81 C4 04 00 00 BF 4D 00 00
C0 BA 0F 00 00 00 E9 B8 FF FF FF FF FF 50 FF 55 08 81 C4 04 00 00 00 SF
C5 C2 04 00 "

It is easy to transform this into a binary, and link with some wrapper code, or even (as we have done) to append a small boot loader and burn a bootable CD.

Even this tiny example shows the power of Coq as an assembler. Ordinary Coq definitions, such as call_cdecl, serve as user-defined macros, here expanding to the calling sequence for a single-argument function that uses the x86 cdecl calling convention. The while and let_simple syntax are “built-in” macros that provide looping and procedural control constructs, hiding the use of scoped labels, branching, and a simple calling convention behind a useful abstraction. Observe how assembly code syntax, familiar to users of assemblers such as MASM, is embedded directly in Coq source. Also note how the assembler itself is executed inside Coq (placing the code at fixed address C0000000), and with a little trickery (the last two lines) the output of the assembler can be sent to the console in hexadecimal format.

1.2 Contributions

Modelling. We show how Coq features such as dependent types, type classes, implicit coercions, indexed inductive types, and user-defined notation combine
to support a clean formalization of machine code – in our case, a subset of x86. Many others have formalized machine code, but our approach has a number of interesting features:

- The representation of machine words and operations is concrete, using tuples of booleans to represent bit vectors, and implementing arithmetic and other operations directly as computable Coq functions, but types are informative, with fine-grained parameterization on word lengths. To prove theorems we exploit the embedding $\mathbb{B}^n \hookrightarrow \mathbb{N}$ of binary representation into the natural numbers and the isomorphism $\mathbb{B}^n \cong \mathbb{Z}_{2^n}$ of binary with with integers modulo $2^n$ in order to exploit the rich collection of lemmas from the ssreflect Mathematical Components library.
- Our abstract, monadic treatment of readers and writers, with separate functional, imperative and logical interpretations, is unusual, and facilitates the reuse of binary format descriptions and proofs of “round-trip” properties.
- The semantics of execution itself is particularly concise, due mainly to our use of monadic syntax and careful factoring of the instruction type and auxiliary definitions.

Assembling. Our approach to writing assembly code inside Coq is novel.

- Our use of Coq notation and implicit coercion features lets us write assembly syntax that can literally be cut-and-paste into a standard assembler.
- We support lexically-scoped labels within assembly code, through a kind of higher-order abstract syntax, and build abstractions such as control structures and procedure calling conventions over the top.
- We have proved a full round-trip property for individual instruction encoding and for the assembler itself.

2 Modelling x86

The x86 architecture and instruction set is notoriously complex: the most recent Intel manual detailing the instruction set alone runs to 1288 pages [3]. Although currently we model only a small subset, even here we must get to grips with rich addressing modes and a variable-length, non-uniform instruction format. In addition to aiming for brevity in our Coq description – in order to facilitate readability and ease of extension – we also desired that the semantics be executable. Running the two instructions from the example in Figure 1.1 produces the following state changes (assuming that the initial value of ESP is D0000000):

```coq
Coq < Compute procStateToString (runFor 0 s).
  = " EIP=C0000000 ESP=D0000000 EBP=00000000 EAX=00000000 EBX=00000000 ...
Coq < Compute procStateToString (runFor 1 s).
  = " EIP=C0000001 ESP=CFFFFFFC EBP=00000000 EAX=00000000 EBX=00000000 ...
Coq < Compute procStateToString (runFor 2 s).
  = " EIP=C0000003 ESP=CFFFFFFC EBP=CFFFFFFC EAX=00000000 EBX=00000000 ...
```
2.1 Bits, bytes and all that

We start with a concrete representation for bit vectors.

Definition BITS n := n.-tuple bool.
Definition BYTE := BITS 8.
Definition WORD := BITS 16.
Definition DWORD := BITS 32.
Definition DWORDorBYTE (dword: bool) := BITS (if dword then 32 else 8).

The \( n \)-tuple type of ssreflect is used to represent \( n \)-bit words concretely and efficiently, and synonyms are defined for bytes, 16-bit words and 32-bit words. The DWORDorBYTE type is used for literals in instructions that have byte and 32-bit word variants, and illustrates well how ‘lightweight’ dependent types are handy for formalizing machine semantics. Syntax is provided for decimal, hexadecimal and binary constants:

Example fortytwo := #42 : BYTE.
Example fortytwo' := #x"2A".
Example fortytwo'' := #b"00101010".

The use of indexing in the type of words supports accurate typing of machine operations such as arithmetic, shifts, and rotates. For example, the “carry-out” of operations such as addition and shift-left is simply expressed as the most significant bit of the result, and full bit-length multiplication produces a result whose size is the sum of the sizes of its inputs:

Fixpoint adcB n carry : BITS n -> BITS n -> BITS n.+1 := ...
Definition dropmsb {n} : BITS n.+1 -> BITS n.
Definition addB {n} (p1 p2: BITS n) := dropmsb (adcB false p1 p2).
Definition fullmulB {n1 n2} : BITS n1 -> BITS n2 -> BITS (n1+n2) := ...
Definition shlB {n} : BITS n -> BITS n.+1 := ...
Definition catB {n1 n2} (p: BITS n1) (p2: BITS n2) : BITS (n2+n1) := ...
Notation "x ## y" := (catB x y) (right associativity, at level 60).

This concrete representation also makes it relatively efficient, which is important to us as we wish to perform bit-level computation inside Coq.

The proofs of useful properties of many operations often proceeds by induction on \( n \). For arithmetic, however, we found it useful to map machine representations onto more abstract mathematical types provided by ssreflect, either embedding \( \text{BITS} n \) into \( \text{nat} \) or using the bijection with \( \mathbb{Z}_{(2^n)} \), the type of integers modulo \( 2^n \). Here, for example, is the proof that addition is associative.

Lemma addBA n : associative (@addB n).
Proof. move => x y z. destruct n; first apply trivialBits.
  apply toZp_inj. rewrite 4!toZp_addB.
  by rewrite addrA. Qed.

The first line of the proof discharges the trivial case of zero bits. Next we use the lemma toZp_inj that states that toZp: \( \text{BITS} n \rightarrow \mathbb{Z}_{(2^n)} \) is injective, in order to transform the goal from \( \text{addB} x (\text{addB} y z) = \text{addB} (\text{addB} x y) z \) to
toZp (addB x (addB y z)) = toZp (addB (x y) z). We then push the embedding inwards using the lemma toZp_addB which expresses that toZp behaves homomorphically with respect to addB and addition on the ring \( \mathbb{Z}_{(2^n)} \). Finally we use the associativity of addition on the target, as provided by the ssreflect library. Many other properties are proved using the same recipe.

### 2.2 Memory

To model memory we use an implementation of finite partial maps whose domain is \( n \)-bit words. We could define this abstractly using \( \text{BITS } n \to \text{option } V \), but as with machine words we prefer something more concrete. To this end we define a variant on tries, with \( \text{NEMAP } n \) representing non-empty maps, and \( \text{PMAP } n \) for possibly-empty maps:

\[
\begin{align*}
\text{Inductive NEPMAP } : \text{nat} & \to \text{Type} := \\
| \text{VAL} & : V \to \text{NEPMAP } 0 \\
| \text{SPLIT} & : \forall n \text{ (lo hi: NEPMAP } n \text{), NEPMAP } n.+1 \\
| \text{LSPLIT} & : \forall n \text{ (lo : NEPMAP } n \text{), NEPMAP } n.+1 \\
| \text{RSPLIT} & : \forall n \text{ (hi : NEPMAP } n \text{), NEPMAP } n.+1.
\end{align*}
\]

\[
\text{Inductive PMAP } n := \\
| \text{PMap} & : \text{NEPMAP } n \to \text{PMAP } n \\
| \text{EmptyPMap} & : \text{PMAP } n.
\]

Definition lookup \( n \) \((m: \text{PMAP } n) (p: \text{BITS } n) : \text{option } V := ...

At each non-leaf level of the trie, the choice of constructor determines whether the left, right, or both subtrees are present. The result of this unusual fussiness is uniqueness of representation: two values of type \( \text{PMAP } n \) are equal in Coq if and only if they are extensionally equal, as expressed by the following lemma.

\[
\text{Lemma extensional_PMAP } n \text{ V (m1 m2: PMAP V n) :} \\
\text{(forall x, lookup m1 x = lookup m2 x) } \to \text{ m1 = m2.}
\]

Memory is then a partial map from 32-bit addresses to bytes, in which the absence of an element indicates that the memory is not mapped, or is inaccessible.

\[
\text{Definition Mem := PMAP BYTE 32.}
\]

In future we plan to refine the model to account for non-writable and non-executable memory.

### 2.3 Monads

In order to abstract a little from the details of execution, decoding, assembling, and so on, we make use of monads. We employ Coq’s type classes for packaging [13], defining a class \text{MonadOps} for syntax and \text{Monad} for the monad laws, along with some useful notation.
Concrete instances of Monad include an error monad and state monad, both of which are used in the semantics of instruction execution. For reading and writing memory, and decoding and encoding of instructions, we define reader and writer monads, which are discussed next.

2.4 Readers and writers

Reading and writing sequences of bytes – or more abstractly, bytes representing some other type of data, such as 32-bit words, or variable-length instructions – pervades our framework. We found it surprisingly difficult to devise an appropriate interface for these operations.

Firstly, there are issues with edge cases. Typically one has a pointer which is advanced as bytes are read or written; but pointer wrap-around must be handled, when the pointer reaches the last addressable byte, and empty sequences must be permitted. To tackle this we introduced a Cursor type, a value of which is either a concrete n-bit pointer, or a value hwm representing the byte just beyond the end of memory.

Inductive Cursor n := mkCursor : BITS n -> Cursor n | hwm.

When reading or writing bytes, the cursor is advanced, but does not wrap around to zero, instead taking on the value hwm.

A second issue was the need for multiple ‘views’ of reading and writing. Sometimes we wish to view reading and writing in a pure, functional style, in which a reader consumes and a writer produces a sequence of bytes. In other situations we want to make reads and writes on our concrete model of memory, for example, when specifying the execution behaviour of instructions. And a third view, used in specifications, interprets readers and writers as predicates on partial states, with sequencing interpreted by separating conjunction.

In order to support these various views of reading and writing, we introduce inductively-defined terms for readers and writers.

Readers. A reader is defined as follows:

Inductive ReaderTm T :=
| readerRetn (x: T)
| readerNext (rd: BYTE -> ReaderTm T)
A reader for type \( T \) either returns a value of type \( T \) immediately, or consumes a single byte and then continues, or asks for the current value of the cursor and continues. This last feature is used in instruction decoding to implement relative addressing for branch instructions. Given appropriate definitions for monad unit and bind operations, we can define \( \text{MonadOps Reader} \) and \( \text{Monad Reader} \) instances, and easily create readers for various types. For example, here is a reader for 32-bit words, in little-endian format.

\[
\begin{align*}
\text{Instance } \text{readDWORD : Reader DWORD} := \\
&\text{let! } b0 = \text{readBYTE;} \\
&\text{let! } b1 = \text{readBYTE;} \\
&\text{let! } b2 = \text{readBYTE;} \\
&\text{let! } b3 = \text{readBYTE;} \\
&\text{retn } (b3 \#\# b2 \#\# b1 \#\# b0).
\end{align*}
\]

Let’s now interpret readers. First, functionally:

\[
\text{Definition runReader } T : \text{Reader } T \to \text{Cursor } 32 \to \text{seq BYTE} \to \\
\text{option (Cursor } 32 \times \text{seq BYTE} \times T).
\]

Given a list of bytes and a starting “position”, the bytes are read sequentially, resulting in a new position (possibly \( \text{hwm} \)), residual bytes, and a value; the value \( \text{None} \) is returned if there are insufficient bytes or if bytes beyond \( \text{hwm} \) are read.

Our second interpretation is imperative and operates on our model of memory \( \text{Mem} \). This time we distinguish between reading beyond the end of memory (\( \text{readerWrap} \)), and reading an unmapped byte (\( \text{readerFail} \)).

\[
\begin{align*}
\text{Inductive readerResult } T := \\
&\text{readerOk } (x : T) \ (q : \text{Cursor } 32) \ | \ \text{readerWrap} \ | \ \text{readerFail}.
\end{align*}
\]

Definition \( \text{readMem } T : \text{Reader } T \to \text{Cursor } 32 \to \text{Mem} \to \text{readerResult } T. \)

Finally, we can give a \emph{logical} view of a reader, expressed as a ‘store predicate’ (\( \text{SPred} \)) which we discuss in detail elsewhere [7].

\[
\begin{align*}
\text{Definition interpReader } T : \text{Reader } T \to \text{Cursor } 32 \to \text{Cursor } 32 \to T \to \text{SPred}.
\end{align*}
\]

Given a reader \( R \) for type \( T \), then \text{interpReader } R p q x holds if the bytes between cursors \( p \) and \( q \) can be ‘read back’ as value \( x \) of type \( T \). We write \( p \to q : x \) for short, if \( R \) is the canonical reader for \( T \).

\textbf{Writers.} For writers, we again define a term syntax with appropriate monadic unit and bind operations; a writer for type \( T \) is then a function from \( T \) to \( \text{WriterTm unit} \).

\[
\begin{align*}
\text{Inductive WriterTm } \{A\} := \\
&\text{writerRetn } (x : A) \\
&\text{writerNext } (b : \text{BYTE}) \ (w : \text{WriterTm}).
\end{align*}
\]
| writerCursor (w: Cursor 32 -> WriterTm) |
| writerFail. |

Class Writer T := getWriterTm: T -> WriterTm unit.

Instance writeBYTE: Writer BYTE :=
    fun b => writerNext b (writerRetn tt).

As with readers, our monadic syntax makes the definition of writers for various types straightforward:

Instance writeDWORD: Writer DWORD := fun d =>
    let: (b3, b2, b1, b0) := split4 8 8 8 8 d in
do! writeBYTE b0; do! writeBYTE b1; do! writeBYTE b2; do! writeBYTE b3;
retn tt.

We can define functional, imperative, and (slightly surprisingly), logical interpretations of terms:

Definition runWriter T :
    Writer T -> Cursor 32 -> T -> option (Cursor 32 * seq BYTE).

Definition writeMem T :
    Writer T -> Cursor 32 -> T -> Mem -> option (Cursor 32 * Mem).

Definition interpWriter T :
    Writer T -> Cursor 32 -> Cursor 32 -> T -> SPred.

To show that a reader correctly decodes anything produced by a writer, we construct an inductively defined simulation relation between them. The relation simrw x p R W means that reader R simulates writer W for the purpose of reading value x starting at position p. Simulating means that they essentially proceed in lock step, except that they are allowed to read the cursor position independently of each other. The last case is a bit technical: it says that a writer that manages to access memory that's out of range is simulated by any reader.

Inductive simrw T (x: T): Cursor 32 -> Reader T -> WriterTm unit -> Prop :=
| simrw_retn p: simrw x p (readerRetn x) (writerRetn tt)
| simrw_next p r b w': simrw x (next p) (r b) w' -> simrw x p (readerNext r) (writerNext b w')
| simrw_rcursor p r' w: simrw x p (r' p) w -> simrw x p (readerCursor r') w
| simrw_wcursor p r w': simrw x p r (w' p) -> simrw x p r (writerCursor w')
| simrw_fail p r: simrw x p r writerFail
| simrw_hwm r b w': simrw x (hwm _) r (writerNext b w').

If for any x and position p a reader R and writer W are related by simrw then they satisfy a round-trip property:

Class Roundtrip X (R: Reader X) (W: Writer X) :=
    roundtrip: forall x p, simrw x p R (W x).

If such a property holds then we can show the following implication in our logic, which can be interpreted as saying “if writing x at p using writer W produces a
sequence of bytes \texttt{bytes} ending at \texttt{q}, then in any state for which the memory from \texttt{p} to \texttt{q} contains \texttt{bytes}, that same memory can be read back using reader \texttt{R} to produce \texttt{x}'.

Lemma pt_roundtrip X (W: Writer X) (R: Reader X) (RT: Roundtrip R W):
forall p q x bytes, runWriter writeNext p x = Some (q, bytes) -> p -- q :-> bytes |-- p -- q :-> x.

2.5 Instructions

The x86 instruction set design is complex, and shows its history. However, some structure can be discerned, and is sufficient to support the definition of a Coq inductive type for instructions that is \textit{total}: every value in the type represents a valid instruction, for which there is a defined encoding, and, almost always, a defined meaning.

Registers. The 32-bit x86 processor has eight general-purpose registers, a flags register (EFLAGS), and an instruction pointer (EIP). (We do not yet model the legacy segment registers, FPU registers or SIMD registers.) We further divide the general-purpose registers into ESP (the stack pointer) and the remainder, because ESP is not allowed to participate in certain addressing modes.

\begin{verbatim}
Inductive NonSPReg := EAX | EBX | ECX | EDX | ESI | EDI | EBP.
Inductive Reg := ESP | nonSPReg :> NonSPReg -> Reg.
Inductive AnyReg := EIP | regToAnyReg :> Reg -> AnyReg.
\end{verbatim}

Addressing modes. The definitions below cover the families of addressing modes used by most instructions.

\begin{verbatim}
Inductive Scale := S1 | S2 | S4 | S8.
Inductive MemSpec := mkMemSpec
  (base: Reg) (indexAndScale: option (NonSPReg*Scale)) (offset: DWORD).
Inductive RegMem := RegMemR :> Reg -> RegMem
  | RegMemM :> MemSpec -> RegMem.
Inductive Src := SrcI :> DWORD -> Src
  | SrcM :> MemSpec -> Src
  | SrcR :> Reg -> Src.
Inductive Tgt := mkTgtI :> DWORD -> Tgt.
Inductive JmpTgt := JmpTgtI :> Tgt -> JmpTgt
  | JmpTgtM :> MemSpec -> JmpTgt
  | JmpTgtR :> Reg -> JmpTgt.
Inductive DstSrc dw := DstSrcRR (dst src: Reg)
  | DstSrcRM (dst: Reg) (src: MemSpec)
  | DstSrcMR (dst: MemSpec) (src: Reg)
  | DstSrcRI (dst: Reg) (c: DWORDorBYTE dw)
  | DstSrcMI (dst: MemSpec) (c: DWORDorBYTE dw).
\end{verbatim}

To take an example, binary operations such as arithmetic and logical operations take two operands, one of which is also used as the destination. The instruction \texttt{add EAX, [EBX + ECX*4 + 14]} makes use of the most complex addressing
mode, indirecting through an address that is computed as the sum of a base register EBX, an index register ECX scaled by 4, and the fixed offset 14. This would be represented by DstSrcRM EAX (mkMemSpec EBX (Some(ECX, S4)) #14).

Instructions. Now let’s tackle the instructions themselves. We abstract just far enough above the binary encoding, but not too far. Sometimes there is more than one way to encode essentially the same instruction, and we do not make this distinction in the data type. Furthermore, the encoding of branches and certain calls uses EIP-relative addressing. Our data type uses absolute addresses uniformly – this makes it slightly easier to formalize the execution semantics, and much easier to support scoped labels and proof rules for control-flow instructions, as code addresses are uniform. Most of the instructions we’ve formalized are shown below.

\[
\text{Inductive BinOp := OP_ADC | OP_ADD | OP_AND | OP_CMP} \\
| OP_OR | OP_SBB | OP_SUB | OP_XOR. \\
\text{Inductive UnaryOp := OP_INC | OP_DEC | OP_NOT | OP_NEG.} \\
\text{Inductive Condition :=} \\
\text{CC_0 | CC_B | CC_Z | CC_BE | CC_S | CC_P | CC_L | CC_LE.} \\
\text{Inductive Instr :=} \\
| UOP (dword: bool) (op: UnaryOp) (dst: RegMem) \\
| BOP (dword: bool) (op: BinOp ) (ds: DstSrc dword) \\
| MOVOP (dword: bool) (ds: DstSrc dword) \\
| MUL (src: RegMem) | IMUL (dst: Reg) (src: RegMem) \\
| LEA (reg: Reg) (src: RegMem) \\
| JCC (cc: Condition) (cv: bool) (tgt: Tgt) \\
| PUSH (src: Src) | POP (dst: RegMem) \\
| CALL (tgt: JmpTgt) | JMP (tgt: JmpTgt) \\
| RET (size: WORD) | ...
\]

2.6 Operational semantics

In essence, the machine is modelled as a state-to-state transition function, a single step consisting of decoding the next instruction and then executing it. The function is partial – because we only model a subset of instructions, and some behaviour is left unspecified – and so we structure it monadically, layering an option monad over a state monad.

Machine state. The machine state splits into three: registers, flags, and memory. Register state is modelled as a (finite) function:

\[
\text{Definition RegState := AnyReg -> DWORD.}
\]

We model flags separately, in order to model the “unspecified” effect that some x86 instructions have on them.

\[
\text{Definition Flag := BITS 5.} \\
\text{Inductive FlagVal := mkFlag} :> \text{bool -> FlagVal | FlagUnspecified.} \\
\text{Definition FlagState := Flag -> FlagVal.}
\]
Definition CF:Flag := #0. Definition PF:Flag := #2.
Definition OF:Flag := #11.

Putting registers, flags, and memory together gives us the processor state:

Record ProcState := mkProcState
{ registers:> RegState; flags:> FlagState; memory:> Mem }.

Instruction decoding. Instruction decoding is implemented as an instance of Reader, making good use of the monadic syntax mixed with simple Coq computation. The fragment here, for example, has just decoded an FE byte:

let! (opx,dst) = readNext;
if opx == #b"000" then retn (UOP false OP_INC dst) else
if opx == #b"001" then retn (UOP false OP_DEC dst)
else retn BADINSTR)

We interpret the reader using readMem, picking up the memory component from the processor state.

Instruction execution. For instruction execution we make significant use of the state monad, reading and writing registers, flags and memory. Through careful use of auxiliary definitions to capture commonality, our formalization is small and easily understood. Here is the interpretation of the RET instruction.

| RET offset =>
  let! oldSP = getRegFromProcState ESP;
  let! IP' = getDWORDFromProcState oldSP;
  do! setRegInProcState ESP (addB (oldSP+#4) (zeroExtend 16 offset));
  setRegInProcState EIP IP'

Putting it together. Given a machine state, the processor (a) decodes the bytes addressed by EIP to determine which instruction to execute; (b) advances EIP to the next instruction, and (c) executes the instruction. All this is captured by the following single-step transition function, written in monadic style.

Definition step : ST unit :=
  let! oldIP = getRegFromProcState EIP;
  let! (instr,newIP) = readFromProcState oldIP;
  do! setRegInProcState EIP newIP;
evalInstr instr.

3 Assembling x86

A particular emphasis of our work on machine code verification is on using Coq as a place to do everything: modelling the machine, writing programs, assembling or compiling programs, and proving properties of programs. Coq’s powerful notation feature makes it possible to write assembly programs, and higher-level language programs, inside Coq itself with no need for external tools.
3.1 Basics of assembly code

Syntax. At its most superficial, assembly code support means nice syntax for the Instr type, which is achieved using Coq’s Notation and Implicit Coercion features. For example, the instruction `add EAX, [EBX + ECX*4 + 14]` considered earlier is valid syntax both in our Coq development and in popular assemblers for x86 such as MASM.

Labels. An important aspect of assembly programs is the ability to define and reference named labels. To this end we define a type program to represent sequences of instructions, label scoping, and label definition:

\[
\text{Inductive program :=}
\]

\[
\mid \text{prog_instr (c: Instr)} \mid \text{prog_skip} \mid \text{prog_seq (p1 p2: program)}
\]

\[
\mid \text{prog_declabel (body: DWORD -> program)}
\]

\[
\mid \text{prog_label (l: DWORD)}
\]

\[
\text{Infix ";;" := prog_seq ...}
\]

\[
\text{Notation "'LOCAL' l ';' p" := (prog_declabel (fun l => p)) ...}
\]

\[
\text{Notation "l ':'" := (prog_label l) ...}
\]

The first three constructors just give us possibly-empty instruction sequences. The prog_declabel constructor and LOCAL notation introduces a new label name l, scoped within p. This use of Coq variables for object-level ‘variables’ (here, labels) is reminiscent of higher-order abstract syntax. Finally prog_label (with familiar colon notation) is a pseudo-instruction whose address the assembler will assign to the label.

Here is an example of its use, in constructing a simple ‘skip over’ conditional.

\[
(* \text{Determine max(r1,r2), leaving result in r1} *)
\]

\[
\text{Definition max (r1 r2: Reg) : program :=}
\]

\[
\text{LOCAL Bigger;}
\]

\[
\text{cmp r1, r2;; jg Bigger;; mov r1, r2;;}
\]

\[
\text{Bigger;:}.
\]

The assembler. In order to turn a program into a sequence of bytes suitable for execution, we must do two things: assign concrete addresses to prog_declabel-bound variables, and encode the instructions themselves.

For encoding, we create an instance of Writer for instructions, with various helper instances for auxiliary types used in instructions. Here is the fragment that deals with three variants of push:

\[
\text{Instance encodeInstr : Writer Instr := fun instr =>}
\]

\[
\text{match instr with}
\]

\[
\mid \text{PUSH (SrcI c) => do! writeNext #x"68"; writeNext c}
\]

\[
\mid \text{PUSH (SrcR r) => writeNext (PUSHPREF ## encReg r)}
\]

\[
\mid \text{PUSH (SrcM src) => do! writeNext #x"FF"; writeNext (#6, RegMemM src)}
\]

As the reader will observe, writeNext is overloaded: in the first case, it is used to write a BYTE and then a DWORD, in the second case, we concatenate a constant.
prefix of five bits onto a three-bit encoding of registers, and in the third case, we use a writer instance for the x86 r/m32 addressing mode.

For label resolution, we use the classic two-pass approach. On the first pass over program, we instantiate prog_declabel with dummy values, encode instructions only to obtain their size, and update a map with addresses for each occurrence of prog_label. Then on the second pass, we instantiate prog_declabel with the calculated addresses and accumulate instruction encodings. Our current scheme assumes that label occurrences (for example, in branches) have fixed length encodings – a scheme that used more efficient variable-length encodings would iterate until label assignments reach a fixed point.

Combining passes, the assembler has the following signature:

Definition assemble_program (offset: DWORD) (p: program) :
  option (seq BYTE) := ...

The assembler will return None if the instructions run off the end of memory.

Assembler correctness. No discussion of an assembler constructed within a proof assistant would be complete without some discussion of proof; and indeed, we have proved that the assembler does its job. First, we can prove that instruction encoding commutes with instruction decoding:

Instance RoundtripInstr : Roundtrip readInstr encodeInstr.

We can then use the logical interpretation of readers to show the following result:

Theorem assem_correct (offset endpos: DWORD) p code:
  assemble_program offset p = Some code ->
  offset -- endpos :-> code |-- offset -- endpos :-> p.

In other words, if a program p assembles at offset to produce a sequence of bytes code ending at endpos, then those bytes when read back through the decoder produce p.

3.2 Macros

A feature of most assemblers – though perhaps little used, now that most assembly code is machine-produced – is the provision of built-in macros and the ability for the programmer to define their own.

Structured control. Built-in macros are used, for example, to build conditionals and loops without going through the pain of declaring labels and branching. In Coq, simple definitions, together with scoped labels and judicious use of notation, make this very easy. Here, for example, is a conditional construct that tests flags according to condition code cond, and executes pthen or pelse accordingly.

Definition ifthenelse cond pthen pelse :=
  LOCAL THEN; LOCAL END;
  JCC cond true THEN;;
  pelse;; jmp END;;
  THEN;;; pthen;;
  END;;.
A while-loop can be defined similarly, this time incorporating test code \( p_{\text{test}} \) that sets the flags appropriately. (See Figure 1.1 for an example of its use.)

\[
\text{Definition while } p_{\text{test}} \text{ cond } \text{body} := \\
\text{LOCAL BODY; LOCAL TEST; } \\
\text{jmp TEST;; } \\
\text{BODY;; body;; } \\
\text{TEST;; p_{\text{test}};; JCC cond true BODY.}
\]

These definitions really shine when proving correctness of machine code programs using our separation logic framework [7], as we can give derived Hoare-style proof rules for the macros.

**Procedures.** We can also devise macros to package up procedures and their calling conventions. Here, for example, is a non-standard ‘leaf’ calling convention in which the return address is simply stored in the register \( EDI \), together with some notation for locally-scoped procedure declarations.

\[
\text{Definition callproc } f := \\
\text{LOCAL iret; mov EDI, iret;; jmp f;; } \\
\text{iret;;.}
\]

\[
\text{Definition defproc } (p: \text{program}) := \\
p;; \text{JMP EDI.}
\]

\[
\text{Notation } \langle \text{letproc} \rangle f ::= \langle \text{p in q} \rangle := \\
\text{(LOCAL skip; LOCAL f; jmp skip;; f;; defproc } p;; \text{skip;; q}) ...
\]

\[
\langle \text{* Multiply EAX by nine, trashing EBX *\rangle \\
\text{Example ex := } \\
\text{letproc tripleEAX := mov EBX, EAX;; shl EAX, 2;; add EAX, EBX } \\
\text{in callproc tripleEAX;; callproc tripleEAX.}
\]

**Parameters and local variables.** Such definitions can be extended to support more complex calling conventions, including \( n \)-ary parameter passing and local variables, with appropriate epilogue and prologue code in the procedure itself. Here is a variant on \( \text{call_cdecl} \) that passes \( n \) parameters.

\[
\text{Fixpoint pushArgs } n (p: \text{program}) : \text{nfun Src } n \text{ program} := \\
\text{if } n \text{ is } n.++1 \text{ return nfun Src } n \text{ program } \\
\text{then fun arg => pushArgs } n (\text{push arg;; } p) \text{ else } p.
\]

\[
\text{Definition call_cdecl_with } (n: \text{nat}) (f: JmpTgt) := \\
pushArgs } n (\text{CALL } f;; \text{add ESP, (n*4)}).
\]

\[
\langle * \text{ Call procedure through EDI with parameters EAX and constant 10. } \text{This will expand to push 10;; push EAX;; CALL EDI;; add ESP, 8 *\rangle \\
\text{Example ex := call_cdecl_with } 2 \text{ EDI EAX 10.}
\]
4 Discussion

There is a long tradition of using proof assistants to formalize processors and verify low-level programs. Notable early work includes the verification in the Boyer-Moore prover of the the Piton assembler for the verified FM8502/9001 microprocessors [9]. For reasons of space, we here discuss just a few more recent pieces of related work.

Mulligan and Sacerdoti Coen [11] have used Matita to formalize the MCS-51 microcontroller and the correctness of an assembler for it. The assembler uses pseudoinstructions for jumps to labels, which may be expanded into different kinds of branch in the object code. Correctness is then presented as a simulation between the operational semantics of object and assembly programs, but the result is conditional on the program (dynamically) not manipulating addresses in ways judged to be ill-behaved (e.g. performing arithmetic on them), and on the correctness of the branch displacement decisions. Our assembler does not currently make branch decisions; we plan to address this at a slightly higher level of abstraction, giving a semantic (rather than syntactic) interpretation to a low-level type system distinguishing pointers from more concrete data.

The targets of the CompCert compiler [8] are comparatively high-level assembly languages for PowerPC, ARM and x86. The operational semantics of these assembly languages are defined over over a C-like memory model (rather than the lower-level array of bytes view we take here), and there are pseudoinstructions for label, allocation and stack operations. CompCert’s treatment of machine integers is less pervasively dependent than ours, exploiting Coq’s module system to provide operations on integers of particular sizes, and using Coq’s arbitrary precision integers, together with proofs that they are within a certain range, rather than raw sequences of bits.

RockSalt [10] is a verified checker for the sandboxing policy used in Google’s Native Client. The verification of RockSalt relies on a Coq model of x86, built using two domain specific languages. One is a regular expression parser, used to decode bitstrings in memory into a inductive datatype of x86 instructions. The abstraction provided by this DSL is similar to that of our reader monads, though non-determinism in the grammars makes establishing the determinism of decoding somewhat involved. The second DSL is a register transfer language, used to define the transition function for decoded instructions, which plays a similar role to the imperative monadic language we use for the same purpose. Our embeddings are shallow, however, whilst those used in RockSalt are deep (i.e. there are inductive datatypes for both grammars and RTL instructions). RockSalt uses CompCert’s libraries for machine integers.

Chlipala has built a model of x86 in Coq for the purpose of verifying OCaml-extractable verifiers for machine code [2]. In this work, the Coq functions work on an assembly-level type of instructions, with binary decoding being delegated to an OCaml function.

Fox and Myreen have formalized the ARMv7 ISA in HOL4 [5]. This is an unusually comprehensive formalization, presented in a monadic style not entirely unlike ours, and has been subjected to testing against real hardware. Myreen has
also built a model of x86 machine code and used it to verify a JIT compiler for a simple language [12]. This model is notable for carefully modelling the x86 instruction cache and treating both encoding and decoding of instructions. There are many similarities between these projects and the work described here, but none have our focus on making it convenient to actually write the machine code that we wish to run and verify entirely within the proof assistant. For producing trustworthy software, proof assistants are the only kind of tool in which all the artefacts in which we are interested (programs, languages, specifications, proofs, . . . ) can coexist and be formally related. Working entirely within a system like Coq thus seems an attractive approach to achieve the highest levels of assurance, and may be more practical than one might at first suppose.

References