Generating & Proving x86 Code in a Proof Assistant

Language

```
Definition allocImp (heapinfo:DWORD)
      (bytes:nat) (fail:DWORD) :=
  mov ESI, heapinfo;;
  mov EDI, [ESI];;
  add EDI, bytes;;
  jc fail;;  (* wrap-around *)
  cmp [ESI+4], EDI;;
  jc fail;;  (* no memory *)
  mov [ESI], EDI.
```

```
Definition e
 (proc(r arg
 const a = 5
 const b = 8
 block k ( x
   (var c =
 if (a==b)
 then goto k (a,a)
 else goto k (b,b))%twiddle.
```

Specification

```
Definition allocSpec n fail inv code :=
  Forall i, Forall j, (
      safe @ (EIP ~= fail ** EDI?) //\\
      safe @ (EIP ~= j ** Exists p,
                EDI ~= p +# n **
                memAny p (p +# n))
    -->>
      safe @ (EIP ~= i ** EDI?))
    @ (ESI? ** OSZCP_Any ** inv)
    <@ (i -- j :-> code).
```

Logic

```
Lemma spec_at_or_and S R1 R2
     {HNeg: AtContra S}:
  S @ (R1 \\// R2) |-- S @ R1 //\\ S @ R2.
Proof.
  rewrite ->land_is_forall, lor_is_exists.
  transitivity (Forall b,
  S @ (if b then R1 else R2)); last first.
  - apply: lforallR => [[|]].
    - by apply lforallL with true.
    - by apply lforallL with false.
  apply: at_ex'.
Qed.
```

Compiler

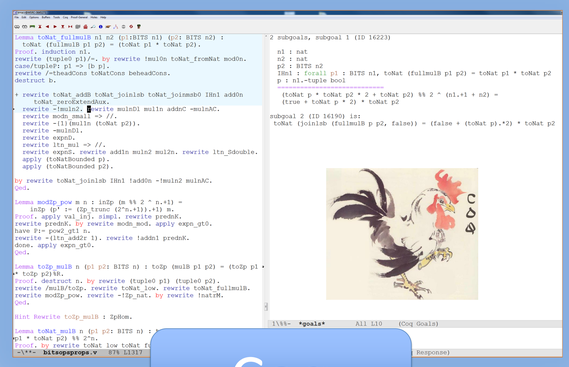
```
| SHIFTOP dword op dst ShiftCountCL =>
  encodeOpcode dword #x"D2" $$
  writeNext (inj op, dst)

| IMUL dst src =>
  writeNext #x"0F" $$
  writeNext #x"AF" $$
  writeNext (inj dst, src)
```

```
| COND x c y
let:noblocks
let:yesblock
let (cc,cv)
    (prefix
mov EAX, (ma
cmp EAX, (makeLocalMemSpec frameReg y);
JCC cc cv (nth #0 cmap (size yesblocks).-1
);
jmp (nth #0 cmap (size noblocks).-1)
```

Proof

```
 (* mov [ESI], EDI *)
specintro. move/eqP => Hcarry0.
subst carry0.
specapply MOV_M0R_rule.
- by ssimpl.
rewrite <-spec_reads_frame.
apply limplValid.
autorewrite with push_at.
 apply: landL2. cancel1.
 rewrite /OSZCP_Any /flagAny /regAny
    /allocInv. ssplits.
```



Coq

Coq is an interactive proof assistant: one can formalize, and have the computer check, arbitrary mathematics. It is also a programming language, with a *very* expressive type system.

Starting with operations on bits and words, we build a Coq model of a subset of the x86 ISA, including decoding and execution.

On top of that, we define languages and compilers, such as a macro-assembler. These execute within Coq and the resulting binaries boot on real hardware.

We also define custom specification languages and program logics in Coq; here a form of Hoare logic for heap data and code pointers. The meaning and correctness of the logics are formally proved right down to the machine model.

The correctness of particular programs can then be proved within Coq. This yields end-to-end correctness with the very highest level of assurance.

Binary

```
"BB 00 80 0B 00 E9 0B 00 00
 43 01 4F FF C3 FF C3 81 FB
 82 E9 FF FF FF BE 00 80 0B
 00 E9 10 00 00 00 8B 06 89
 00 00 81 C7 04 00 00 00 81
 0F 82 E4 FF FF FF B9 14 00
 00 00 BF D1 06 30 00 C1 E2
 02 03 FA C1 EA 07 C6 04 4F
 00 00 0F 84 07 00 00 00 FF
 00 B9 00 00 00 00 81 FA 31
 07 00 00 00 FF C2 E9 05 00
```


Andrew Kennedy (akenn)
Nick Benton (nick)
MSR Cambridge

x86 Architecture

```
Inductive NonSPReg := | EAX | EBX | ECX |
EDX | ESI | EDI | EBP.
(* General purpose registers,
   including ESP *)
Inductive Reg :=
| nonSPReg :> NonSPReg -> Reg
| ESP.
(* All registers, including EIP
   but excluding EFL *)
Inductive AnyReg :=
| regToAnyReg :> Reg -> AnyReg
| EIP.
```

x86 Semantics

```
| MUL src =>
  let! v1 = getRegFromProcState EAX;
  let! v2 = evalRegMem src;
  let res := fullmulB v1 v2 in
  let cfof := high 32 res == #0 in
  do! setRegInProcState EAX (low 32 res);
  do! setRegInProcState EDX (high 32 res);
  do! updateFlagInProcState CF cfof;
  do! updateFlagInProcState OF cfof;
  do! forgetFlagInProcState SF;
  do! forgetFlagInProcState PF;
  forgetFlagInProcState ZF
```