Reasoning About Y86 Machine Code

J Strother Moore
Department of Computer Science
University of Texas at Austin

What is Y86?

a simple machine code model distantly related to X86

introduced by Randal E. Bryant and David R. O'Hallaron in *Computer Systems, A Programmer's Perspective*

ACL2 model (including an assembler) developed by Warren Hunt

Demo 1

. . . in which I show some example Y86 assembly code programs, a Y86 state, and give a very brief tour of the definition.

Controlling Complexity

Greve's *wormhole abstraction* implemented with *effects*

iY86, an *intensional* subset of Y86

re-usable *generic* theorems

macros for reasoning about *straight code sequences*, *loops*, and *procedure calls*

The Effects Relation

```
(-=> s (effects (:eax 5)
                (:edx (+ 1 (g :eax s)))))
```

means

"the *important aspects* of state `s` are that `:eax` is 5 and `:edx` is 1 + `:eax`"

Formally

```
(defun -=> (s lst) (equal s (s* lst s)))

(s* (list (list key val) ...) s)
= (s key val (s* ... s))
```

Thus, if we know

```
(-=> (y86 s n) (effects ...))
```

then we can replace all occurrences of `(y86 s n)` by

```
(s* (effects ...)
    (y86 s n))
```

Thus, if we know

```
(-=> (y86 s n) (effects ...))
```

then we can replace all occurrences of `(y86 s n)` by

```
(s* (effects ...)
    (y86 s n))
```

Thus, if we know

```
(-=> (y86 s n) (effects ...))
```

then we can replace all occurrences of `(y86 s n)` by

```
(s* (effects ...)
    (etc s n))
```

After this transformation, all we know about `(y86 s n)` is its *effects*.

Proving

```
(-=> (y86 s n) (effects ...))
```

can be done by symbolically running y86 and then just checking that the important effects are as listed. Everything else "is what it is."

(That's Greve's characterization of "wormhole abstraction.")

Intentional Effects

`Y86` collects lots of *extensional* data (e.g., cache behavior, etc.). This data is irrelevant to computations.

If the effects of a computation are purely intentional, `Y86` can be replaced by the simpler `iY86`.

Demo 2

. . . in which I show the relation between the two machines.

Sequential Code

To deal with long code sequences, I *snorkel* the clock.

```
(y86 s 5007)
```

=

```
(y86 (y86 s 200) 4807)
```

Sequential Code

Recall `big1`, iterates 1000 times, incrementing mem[8+%esp]:

```
(irmovl 50000 %esp)    ;;; %esp <- 50000
(irmovl 1000 %ebx)     ;;; %ebx <- 1000
(irmovl 1 %ecx)        ;;; %ecx <- 1
(irmovl 0 %edx)
(subl   %ecx %edx)     ;;; %edx <- (%edx - 1)

big1-loop
...
```

Sequential Code

Recall `big1`, iterates 1000 times, increments mem[8+%esp]:

```
...
big1-loop
(mrmovl 8(%esp) %eax)
(addl %ecx %eax)
(rmmovl %eax 8(%esp)) ;;; mem[50008] <- 1+mem[50008]
(addl %edx %ebx)      ;;; %ebx <- %ebx - 1
(jg big1-loop)        ;;; jump to big1-loop if %ebx > 0
big1-halt
(halt)
```

Demo 3

. . . in which I show how we can reason about long "straight line code" with a special macro by proving `big1`'s effects are

```
(effects (:mem (append
                 (xtr *prog-lo* *prog-hi* (g :mem s))
                 (list (list 50008 (+ 1000 (r32 50008 (g
         (:eip *big1-halt*))
```

by running through the loop 1000 times.

Procedure Call/Return

I verify procedure call/return with standard sequential code techniques: just simulate through the call/return prefix and postfix (which manage the stack, saved registers, and pc).

Loops

But to verify loops I have developed a special macro based on generic theorems.

The key idea is to use sequential code techniques to verify that the loop body has certain effects and then relate the iterated loop effects to the single-pass effects without dealing with code.

The Generic Theorems

`(hyp s)` - the hypothesis of the desired theorem – except you should omit `(y86-guard s)` because that is given to you "for free." The hyp should insure that `:eip` is at the top of the loop.

`(beta s)` - the effects of executing the loop to completion – should specify an `:eip` outside the loop.

`(test s)` - true if we are to exit the loop; however, it is assumed that we will still make one pass through it on our way to the exit because that is so common on the y86 (the code generally contains a conditional-jump at the bottom of a loop).

`(k s)` - the number of steps in the next pass through the loop. Often this is a constant but might be dependent on data in `s`.

`(m s)` - a measure that decreases every time we go through the loop. You may assume both the guard and `hyp` when you write `m`. Thus, a good `m` might be just `(g :edx s)`.

`(alpha s)` - important effects of one pass through the loop; may leave `:eip` back at the top or outside the loop.

Demo 4

. . . in which we explore the constraints on these functions and what those constraints allow us to prove about loops.

A Callable Procedure with a Loop

The `iterative` program sums the numbers from %eax down to 0.

```
(pushl %ebp)                ;;; save registers
(rrmovl %esp %ebp)          ;;; and stack
(pushl %eax)
(pushl %ebx)
(pushl %ecx)
(pushl %edx)
```

```
iterative-start
(xorl    %ebx %ebx)     ;;; %ebx <- 0
(irmovl  -1   %ecx)     ;;; %ecx <- -1

iterative-loop
(addl    %eax %ebx)     ;;; %ebx <- %eax + %ebx
(addl    %ecx %eax)     ;;; %eax <- %eax - 1
(jg iterative-loop)     ;;; jump iterative-loop [if n>0]

iterative-finish
(rmmovl %ebx 100(%edi));;; mem[%edi+100] <- %ebx
```

```
iterative-exit              ;;; restore registers
(mrmovl -16(%ebp) %edx)     ;;; and stack
(mrmovl -12(%ebp) %ecx)
(mrmovl  -8(%ebp) %ebx)
(mrmovl  -4(%ebp) %eax)
(rrmovl %ebp %esp)
(popl %ebp)
(ret)
```

Demo 5

. . . in which we demonstrate the macro for reasoning about loops and then use our sequential macro to reason about procedure call/return through a loop.

Recursive Procedures

`(rhyp s)` - the pre-condition of called routine (given a well-formed state `s`); this must include the requirements on available stack space and that the next instruction is a call of the routine in question!

`(rbeta s)` - the important effects of executing the routine to completion, including that `eip` has been advanced by 5!

`(rtest s)` - true if we are in the base case

`(rkb s)` - the number of steps from the call, through the base case, to the `ret`

`(rkc s)` - the number of steps from the outer call to the inner call.

`(rkd s)` - the number of steps from the

inner recursive call to the `ret`. Note that it is a function of the input state, not the state after the recursive call.

`(rm s)` - a measure that decreases every we see a call. You may assume both the guard and `rhyp` when you write `rm`. Thus, a good `rm` might be just `(g :eax s)`.

Demo 6

. . . in which I show what the constraints on these functions are and what can be derived from them.

Recursive Sum

```
(pushl %ebp)              ;;; Standard entry
(rrmovl %esp %ebp)
(pushl %eax)
(pushl %ebx)
(pushl %ecx)
(pushl %edx)

recursive-body
(xorl    %ebx %ebx)     ;;; %ebx <- 0
(addl    %ebx %eax)     ;;; %eax <- %eax + 0 (set flags)
(jg recursive-pre-step)
```

```
recursive-base
(rmmovl %ebx 100(%edi));;; mem[%edi+100] <- 0
(jmp recursive-exit)

recursive-pre-step
(pushl %eax)              ;;; save local
(irmovl -1 %ebx)
(addl %ebx %eax)          ;;; %eax <- %eax-1

recursive-call
(call recursive)          ;;; mem[%edi+100] <- recursive(‘
```

```
recursive-post-step
(popl %eax)                ;;; restore local
(mrmovl 100(%edi) %ebx);;; get result
(addl %eax %ebx)           ;;; %ebx=%eax+%ebx
(rmmovl %ebx 100(%edi));;; mem[%edi+100] <- answer
recursive-exit
(mrmovl -16(%ebp) %edx);;; Standard exit
(mrmovl -12(%ebp) %ecx)
(mrmovl  -8(%ebp) %ebx)
(mrmovl  -4(%ebp) %eax)
(rrmovl %ebp %esp)
(popl %ebp)
(ret)
```

Demo 7

. . . in which I demonstrate the macro for proving recursive procedures.