The Y86 – A Technology Driver

Warren A. Hunt, Jr. and Matt Kaufmann

February, 2012

Computer Science Department
University of Texas
1 University Way, M/S C0500
Austin, TX 78712-0233

{hunt,kaufmann}@cs.utexas.edu
TEL: +1 512 471 {9748, 9780}
FAX: +1 512 471 8885

Outline

Introduction

Attempting to develop a machine-code verification environment.

- We are developing ISA models suitable for code analysis.
- We are developing a proof environment for ISA code.
- Our work is based on commercial-sized problems.

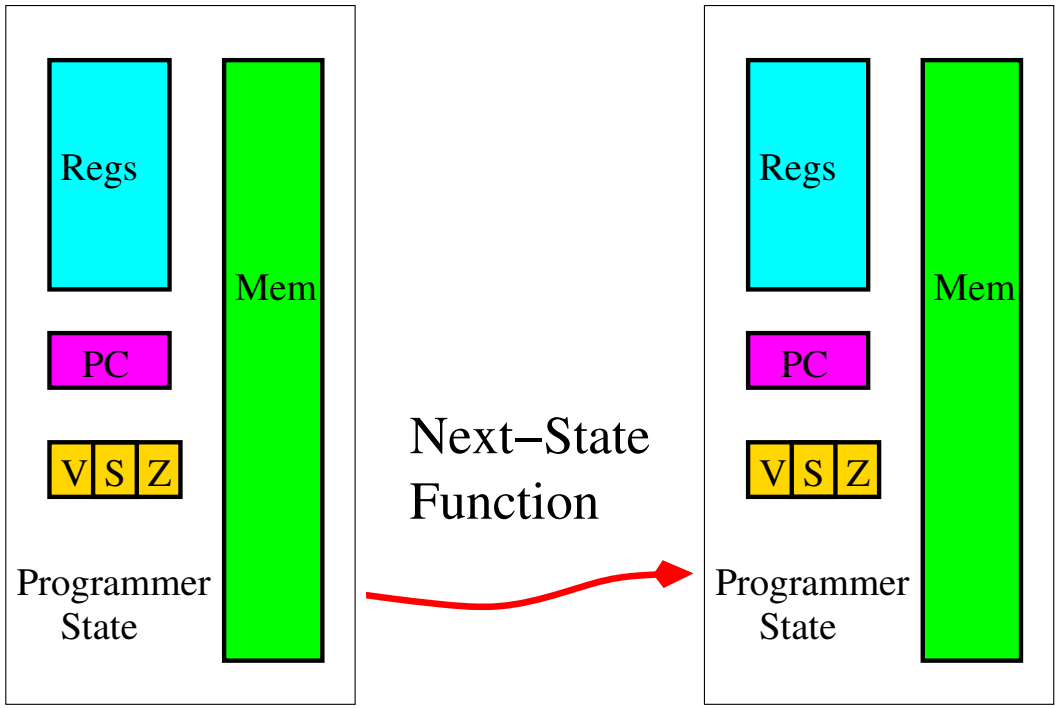
Overall goals:

- Accurate simulation environment for binary code investigation.
- First stage: ability to run FreeBSD/Linux binaries.
- Second stage: ability to boot and run FreeBSD/Linux.
- Provide code analysis capabilities for X86-based systems.

Technology Driver: The Y86

We use our Y86 model as a technology driver.

- Our Y86 model is used by 100+ students in my architecture class
- We are using it as a basis for our hypervisor study project
- We are using it to develop various microprocessor memory models



Y86 Processor Modeling

```
(defund y86 (x86-32 n)
  (declare (xargs :guard (and (natp n)
                              (x86-32p x86-32))
                  :measure (acl2-count n)))
  (if (mbe :logic (zp n) :exec (= n 0))
      x86-32
    (if (ms x86-32)
        x86-32
      (let ((x86-32 (y86-step x86-32)))
        (y86 x86-32 (1- n))))))
```

Y86 Processor Modeling – NOP

```
(defund y86-nop (x86-32)
  (b* ((pc (eip x86-32))

       ;; Memory Probe
       ((if (< *2^32-2* pc))
        (update-ms (list :at-location pc
                         :instruction 'nop
                         :memory-probe nil)
                   x86-32))

       ;; Update PC
       (x86-32 (update-eip (+ pc 1) x86-32)))
    x86-32))
```

Y86 Processor Modeling – Ra + Rb

```
(defund y86-rA-+-rB-to-rB (x86-32)
  (b*
   (;; Get program counter
    (pc (eip x86-32))
    ;; Memory Probe
    ((if (< *2^32-3* pc))
     (update-ms
      (list :at-location pc
            :instruction 'addl
            :memory-probe nil) x86-32))
    ;; Decode registers
    (rA-rB (rm08 (+ pc 1) x86-32))
    (rB (n03 rA-rB))
    (rA (n03 (ash rA-rB -4)))
    (b3 (n01 (ash rA-rB -3)))
    (b7 (n01 (ash rA-rB -7)))
    ;; Register decoding problem?
    ((if (= (logior b3 b7) 1))
     (update-ms
      (list :at-location pc
            :instruction 'addl
            :b7 b7 :b3 b3
            :ra rA :rb rB) x86-32))
    ;; Get register values
    (rA-val (rgfi rA x86-32))
    (rB-val (rgfi rB x86-32))
    ;; Calculate result
    (result (n32+ rA-val rB-val))
    ;; Calculate flags
    (zf  (if (= result 0) 1 0))
    (sfr (< result (expt 2 31)))
    (sf  (if sfr 0 1))
    (sfa (< rA-val (expt 2 31)))
    (sfb (< rB-val (expt 2 31)))
    (of  (if (eq sfa sfb)
             (if (eq sfr sfa) 0 1)
           0))
    ;; Store results
    (x86-32 (update-rgfi rB result x86-32))
    (x86-32 (y86-ALU-results-store-flgs
             zf sf of x86-32))
    ;; Update PC
    (x86-32 (update-eip (+ pc 2) x86-32)))
    x86-32))
```

Y86 Processor Modeling – POP

```
(defund y86-popl (x86-32)
  (b*  ;; Memory probe and decoding check removed!!!
   ((pc (eip x86-32))

    ;; Decode registers
    (rA-rB (rm08 (+ pc 1) x86-32))
    (rA (n03 (ash rA-rB -4)))
    (b3-b0 (n04 rA-rB))
    (b7 (n01 (ash rA-rB -7)))

    ;; Ordering is critical when :esp is
    ;; target, if POPL needs to overwrite
    ;; the contents of the :esp register.
    (esp (rgfi *mr-esp* x86-32))
    (esp+4 (n32+ esp 4))
    (MemAtStackPt (rm32 esp x86-32))
    (x86-32 (update-rgfi *mr-esp* esp+4 x86-32))
    (x86-32 (update-rgfi rA MemAtStackPt x86-32))

    ;; Update PC
    (x86-32 (update-eip (+ pc 2) x86-32)))
     x86-32))
```

Popcount Specification

Lisp has a built-in population count function: LOGCOUNT.

In C, it might look like:

```
int popcount ( int v ) {
  return (( v == 0 ) ? 0 : (v & 1) + popcount( v / 2 )); }
```

Thus, we would like to verify a similar program.

Popcount Program

Here is a population count program in Y86 assembler.

```
(defconst *popcount-source*

  ;; Register Usage:
  ;;   %eax -- evolving count
  ;;   %ebx -- input argument, which is erased as it is counted
  ;;   %ecx -- tmp
  ;;   %edx -- mask, starts as 1 and is shifted left each iteration
  ;;   %esi -- constant 1

  '(popcount

    ;; Subroutine setup
    (pushl  %ebp)              ;   0: Save superior frame pointer
    (rrmovl %esp %ebp)         ;   2: Set frame pointer
    (pushl  %ebx)              ;   4: Save callee-save register
    (pushl  %esi)              ;   6: Save callee-save register

    (mrmovl 8(%ebp) %ebx)      ;   8: Get <v>
    (xorl   %eax %eax)         ;  14: %eax <- 0
    (irmovl 1 %esi)            ;  16: %edx <- 1
    (rrmovl %esi %edx)         ;  22: %esi <- 1
```

Popcount Program

```
loop

(rrmovl %ebx %ecx)        ;  24: Evolving <v>
(andl   %edx %ecx)        ;  26: Bit a 1?
(je     move_mask)        ;  28  Jump if bit is zero

(xorl   %edx %ebx)        ;  33: Erase bit just counted
(addl   %esi %eax)        ;  35: Add 1 to the count

move_mask

(addl   %edx %edx)        ;  37: Shift mask left
(andl   %ebx %ebx)        ;  39: Anything left to count?
(jne    loop)             ;  41: If so, keep counting

popcount_leave

(popl   %esi)             ;  46: Restore callee-save register
(popl   %ebx)             ;  48: Restore callee-save register
(rrmovl %ebp %esp)        ;  50: Restore stack pointer
(popl   %ebp)             ;  52: Restore previous frame pointer
(ret)                     ;  54: Subroutine return
```

Popcount Program Proof Request

```
(def-gl-thm y86-popcount-correct-reduced-symsim
  :hyp (n32p n)
  :concl (let* ((start-eip (cdr (assoc-eq 'call-popcount
                                          *popcount-symbol-table*)))
                (halt-eip (cdr (assoc-eq 'halt-of-main
                                         *popcount-symbol-table*)))
                (esp 8192)
                (x86-32 (popcount-init-x86-32 n esp start-eip))
                (count 300)
                (x86-32 (y86 x86-32 count)))
           (and (equal (rgfi *mr-eax* x86-32)
                       (logcount n))
                (equal (eip x86-32)
                       halt-eip)))
  :g-bindings
  `((n   (:g-number ,(gl-int 0 1 33)))
    (esp (:g-number ,(gl-int 33 1 33))))
  :rule-classes nil)
```

Fibonacci Specification

```
(defun fib (x)
  (declare (xargs :guard (natp x)))
  (mbe :logic
       (if (zp x)
           0
         (if (= x 1)
             1
           (+ (fib (- x 2)) (fib (- x 1)))))
       :exec
       (if (< x 2)
           x
         (+ (fib (- x 2)) (fib (- x 1))))))
```

Aside: ACL2 functions can be memoized.

```
(memoize 'fib :condition '(< 40 x))
```

Equivalent Function Proof Statement

```
(defun f1 (fx-1 fx n-more)
  (declare (xargs :guard (and (natp fx-1)
                              (natp fx) (natp n-more))))
  (if (zp n-more)
      fx
    (f1 fx (+ fx-1 fx) (1- n-more))))

(defun fib2 (x)
  (declare (xargs :guard (natp x)))
  (if (zp x)
      x
    (f1 0 1 (1- x))))

(defthm fib2-is-fib
  (implies (natp x)
           (equal (fib2 x) (fib x))))
```

Fibonacci Program

```
(defconst *fib-source*
  '(fib
    ;; Subroutine setup
    (pushl  %ebp)          ;   0: Save superior frame pointer
    (rrmovl %esp %ebp)     ;   2: Set frame pointer
    (pushl  %ebx)          ;   4: Save callee-save registers on stack
    (pushl  %esi)          ;   6:

    (mrmovl 8(%ebp) %ebx) ;   8: Get <N>

    ;; Zero test

    (xorl   %eax %eax)     ;  14: %eax := 0
    (andl   %ebx %ebx)     ;  16: Set flags
    (jle    fib_leave)     ;  18: Return 0, if <N> <= 0

    ;; One test

    (irmovl 1 %eax)        ;  23: %eax := 1
    (rrmovl %ebx %ecx)     ;  29: %ecx := <N>
    (subl   %eax %ecx)     ;  31: %ecx := <N> - 1
    (je     fib_leave)     ;  33: Return 1, if <N> == 0
```

Fibonacci Program

```
;; Push (- N 1) on stack for recursive FIB call

(pushl  %ecx)          ;  38: Push (<N> - 1)
fib-1
(call   fib)           ;  40: Recursively call fib(<N> - 1)
(popl   %ecx)          ;  45: Restore stack pointer
(rrmovl %eax %esi)     ;  47: Save fib(<N> - 1)

(irmovl 2 %ecx)        ;  49:
(subl   %ecx %ebx)     ;  55: <N> - 2

;; Push (- N 2) on stack for recursive FIB call

(pushl  %ebx)          ;  57: Push (<N> - 2)
fib-2
(call   fib)           ;  59: Recursively call fib(<N> - 2)
(popl   %ecx)          ;  64: Restore stack pointer

(addl   %esi %eax)     ;  66: fib(<N> - 2) + fib(<N> - 1)
```

Fibonacci Program

```
;; Subroutine leave

fib_leave

(popl   %esi)          ;  68: Restore callee-save register
(popl   %ebx)          ;  70: Restore callee-save register
(rrmovl %ebp %esp)     ;  72: Restore stack pointer
(popl   %ebp)          ;  74: Restore previous frame pointer
(ret)                  ;  76: Subroutine return

end-of-code
```

Main Y86 Program, Used to Call FIB

```
;; Main program

(align   16)            ;  80: Align to 16-byte address
main                    ;  80: "main" program
(irmovl  stack %esp)    ;  80: Initialize stack pointer (%esp)
(rrmovl  %esp %ebp)     ;  86: Initialize frame pointer (%ebp)
(irmovl  6 %eax)        ;  88: <N>:  fibonacci( <N> )

(pushl   %eax)          ;  94: Push argument on stack
call-fib
(call    fib)           ;  96: Call Fibonacci subroutine
return-from-fib

(popl    %ebx)          ; 101: Restore local stack position
(halt)                  ; 103: Halt

;; Stack

(pos 8192)              ; 8192: Assemble position
stack                   ; 8192: Thus, "stack" has value 8192
))
```

Fibonacci Program Step Count

```
(defun fib-count (n)

; Return the number of steps taken by y86,
; starting at a call of FIB and ending just
; after corresponding (ret) in the FIB routine.

  (declare (xargs :guard (natp n)
                  :ruler-extenders :all))
  (1+                     ; (for the call instr)
   (cond ((zp n) 13)      ; 8 (prelude) + 5 (postlude, to fib_leave)
         ((eql n 1) 17) ; 8 + 4 (extra prelude when N = 1) + 5
         (t (+ 13         ; 8 (prelude)
               (fib-count (- n 1))
               5
               (fib-count (- n 2))
               7)))))
```

Fibonacci Initial State

```
(defun fib-init-x86-32 (n esp eip)

; N is our formal, esp is the stack pointer
; just before (call fib), and eip position of
; the (call fib) instruction.  It's important
; that addresses from *fib-binary* don't include
; esp, and in fact there's sufficient separation
; to let the stack grow as fib is executed
; without smashing the fib code.

  (declare (xargs :guard (and (n32p n)
                              (n32p esp)
                              (n32p eip))))
  (init-y86-state nil
                  eip
                  `((:esp . ,esp))
                  nil
                  *fib-binary*
                  (wm32 esp n (create-x86-32)) ; n is on the top of the stack
                  ))
```

Y86 Poised to Execute FIB

```
(defun poised-at-fib-n (n x86-32)
  (declare (xargs :guard (and (n32p n) (x86-32p x86-32))))
  (let ((esp (rgfi *mr-esp* x86-32)))
    (and (n32p (+ 3 esp))
         ;; Call has not yet taken place (next step is the call)
         (equal n (rm32 esp x86-32))
         ;; We check that the stack necessary won't overwrite the
         ;; code.  The nesting of calls is at most n, and for each
         ;; stack frame we push four doublewords.
         (<= (cdr (assoc-eq 'end-of-code ; just past the return
                            *fib-symbol-table*))
             (- esp ; subtract max number of bytes to be pushed
                (fib-stack-max-bytes n))))))

(defun poised-at-fib-base (eip x86-32)
  (declare (xargs :guard (and (x86-32p x86-32) (n32p eip))))
  (and (mem-segment-p *fib-binary* x86-32)
       (equal (eip x86-32) eip)))

(defun poised-at-fib (n eip x86-32)
  (declare (xargs :guard (and (n32p n) (n32p eip) (x86-32p x86-32))))
  (and (poised-at-fib-base eip x86-32)
       (poised-at-fib-n n x86-32)))
```

Minimum Requirement to Execute FIB

```
(defun reduce-fib (x86-32)
  (declare (xargs :guard (x86-32p x86-32)))
  (let ((esp (rgfi *mr-esp* x86-32)))
    (fib-init-x86-32 (rm32 esp x86-32)
                     esp
                     (cdr (assoc-eq 'call-fib
                                    *fib-symbol-table*)))))
```

Fibonacci Proof Statement

```
(defthm y86-fib-correct-up-to-3-reduced
  (let ((eip (cdr (assoc-eq 'call-fib *fib-symbol-table*)))
        (esp (rgfi *mr-esp* x86-32)))
    (implies (and (natp n)
                  (< n 3)
                  (n32p esp)
                  (n32p (+ esp 3))
                  (f-stack-okp n esp)
                  (x86-32p x86-32)
                  (poised-at-fib n eip x86-32))

             (let* ((x86-32 (reduce-fib x86-32))
                    (x86-32 (y86 x86-32 (fib-count n))))

               (and (equal (rgfi *mr-eax* x86-32)
                           (fib n))

                    (equal (eip x86-32)
                           (cdr (assoc-eq 'return-from-fib
                                          *fib-symbol-table*)))))))
```

Conclusion

We continue to expand our modeling and analysis capabilities.

- We are attempting to use our symbolic-simulation capability to verify programs.
- We want to specify a executable subset a commercial ISA that is suitable for analyzing programs.
  - We are building memory models
  - We are developing a co-simulation mechanism for model validation
  - Such a model can be used as a built-to and a compile-to specification
  - Such a model can symbolically execute code
  - Such a model can be used to safely explore all manner of malware
- We wish to take advantage of asynchronous computation where appropriate.

We perform all of our work in an environment where we can prove or disprove theorems about our models.