An Update on Oracle's Use of ACL2

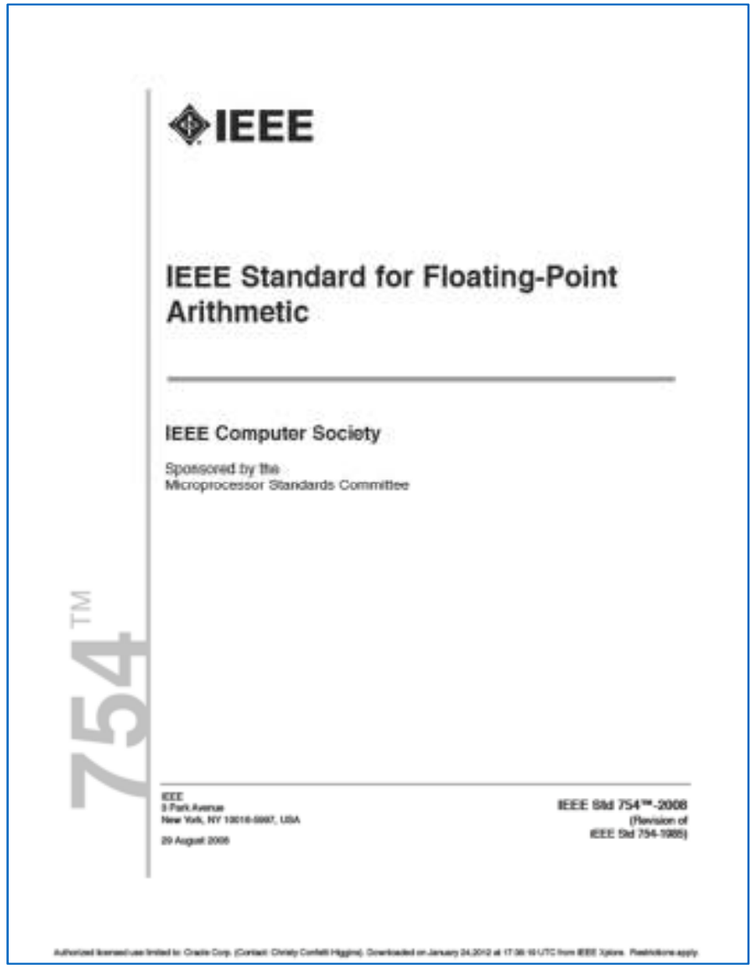
May 22, 2017

Current In-House Theorem Proving Team

- Andrew Brock, Jo Ebergen, Keshav Kini, Dmitry Nadezhin, David Rager

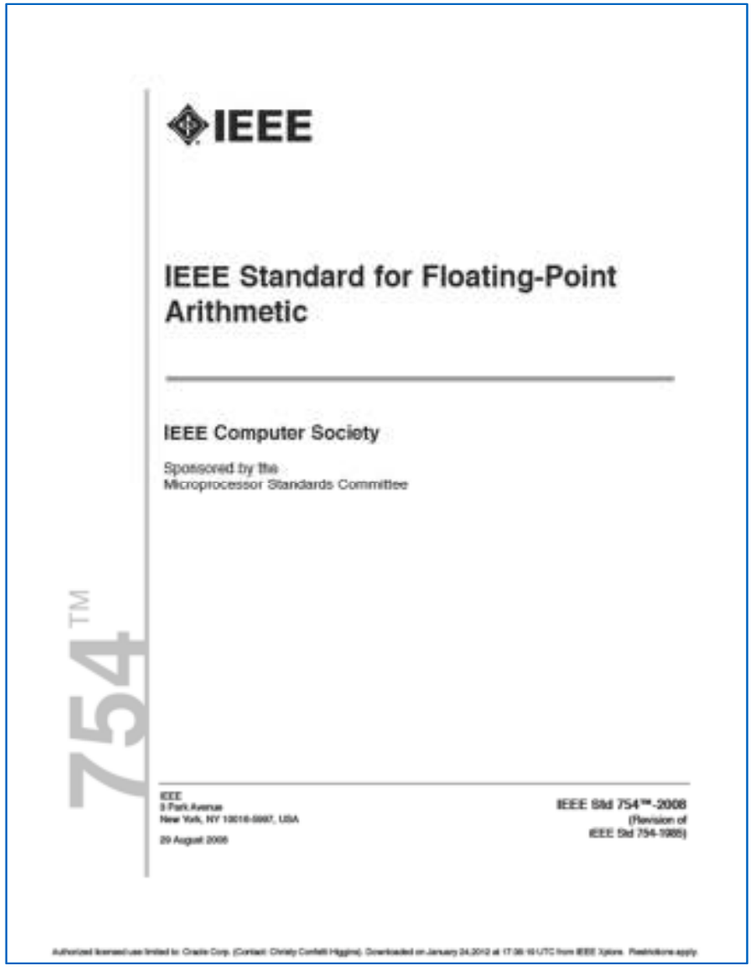
Outline

- *Floating-point*
- Control logic verification, a case of starvation
- Invariant extraction



On Floating-Point

- Still doing some floating-point work
- Proofs are required before tape out
- Still finding and verifying performance improvements
- Not the focus of this talk


IEEE

IEEE Standard for Floating-Point Arithmetic

IEEE Computer Society

Sponsored by the
Microprocessor Standards Committee

754™

IEEE
3 Park Avenue
New York, NY 10016-5997, USA
29 August 2008

IEEE Std 754™-2008
(Revision of
IEEE Std 754-1985)

Outline

- Floating-point
- *Control logic verification, a case of starvation*
- Invariant extraction

A Control Logic Problem to Solve

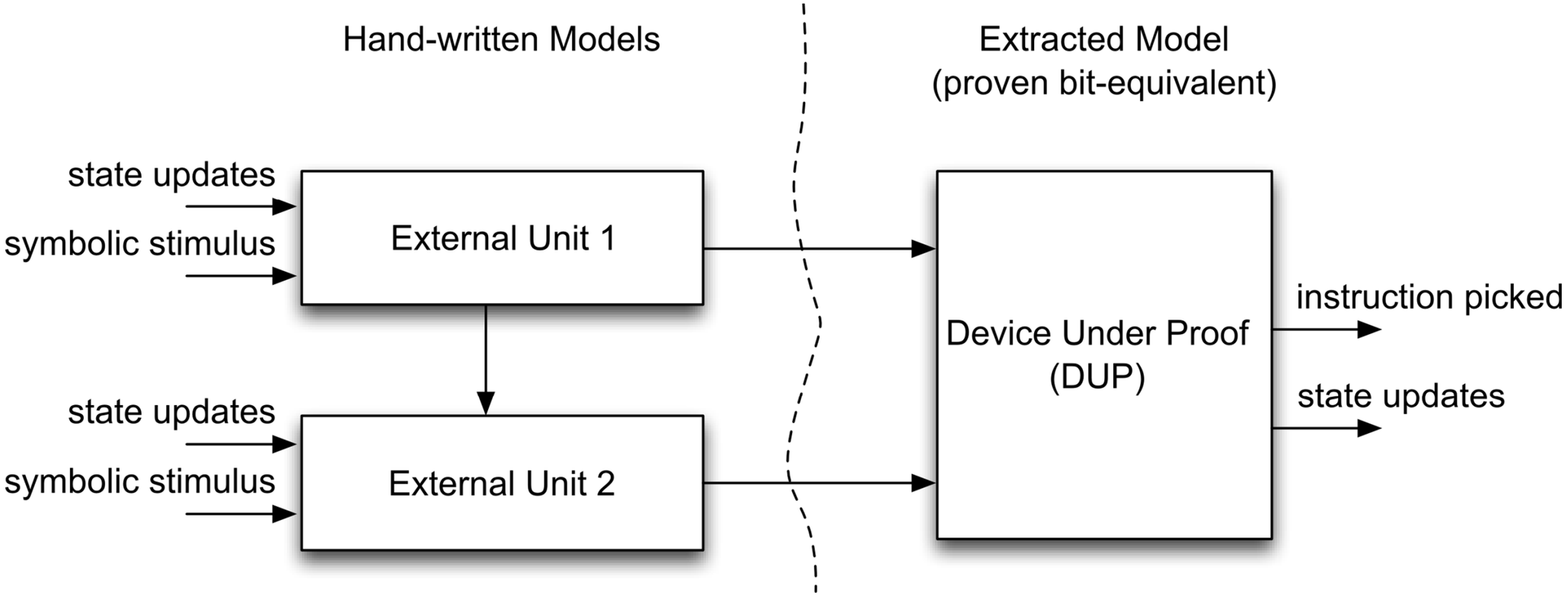
- Can we prove that a certain class of instructions will progress?
  - Alternatively: can we construct an execution that starves a particular instruction?
- Hardware implementation challenges:
  - 8-way hardware threading
  - Out of order execution
  - Instructions that take a long time to execute
  - Power management
  - …

A Case of Starvation

- <Picture elided>

- Difficult to imagine problematic executions; need help from tool (ACL2 stack)

Applicable Models



Winds in Our Favor

- Small and bounded durations of concern
  - Typically less than 100 cycles
- Models for Device Under Proof (DUP) and devices surrounding the DUP can be written in GL-friendly primitives (logand, loghead, etc.)
- Model for DUP is proven bit-for-bit equivalent to the Verilog RTL
- Hand-written external models are much simpler than an extracted model
  - Can be easier to play with external modules and see if it breaks proofs
  - No concrete simulation necessary to test ideas
- ACL2, GL, and SAT solvers are remarkably powerful

Winds Against Us

- Writing the intermediate model for the DUP that's at a higher level than SVEX takes time
  - Already automated part of this but still work to do
- Hand-written models for the external units are vulnerable to typos, connection design flaws, and other mistakes

End Result

- We constructed an execution path for concurrently executing instructions that resulted in indefinitely preventing a particular instruction from executing
  - ACL2 + GL + SAT (glucose 3.0) generated the counterexample
- Represented a bug in the design

Benefits of a Symbolic Trajectory Evaluation-like Approach (defsvtv)

- Fast to get up and running
- If the proofs go through automatically, don't have to think about what's happening in each cycle nearly as much
  - Can speedup process by 2-5x
- Great for datapath verification, so long it's okay to narrow the scope of the proof by tying off some control logic signals
- Svtv-debug timing diagrams easily accessible and intuitive

- Doesn't seem as good for verifying control logic….

Outline

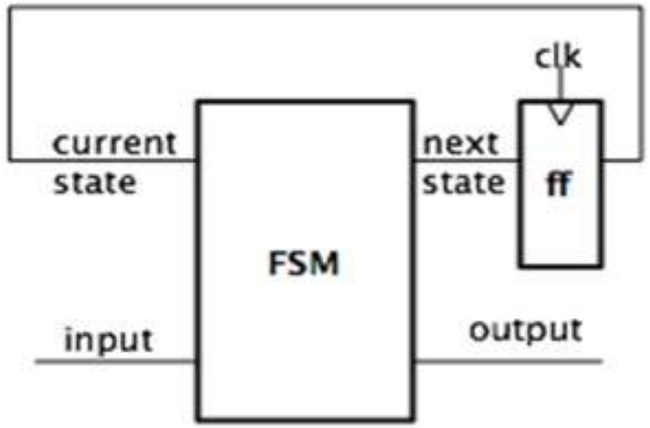
- Floating-point
- Control logic verification, a case of starvation
- *Invariant extraction*

Two Classes of Invariants (for today only)

- Single-cycle invariant
- Multi-cycle invariant

Single-Cycle Invariant

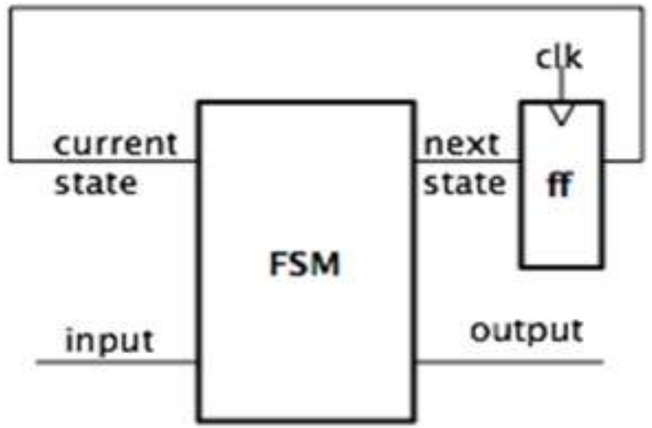
- Many ways to extract, we use a single defsvtv simulation cycle
  - Inputs (and overrides) come in
  - Read outputs one simulation cycle later
  - Make simple functional statement about outputs (and internals) in terms of inputs



- E.g., “this internal wire represents a partial product of inputs A and B”
- E.g., “this output represents the addition of some internal wires”

Converting a Single-Cycle Invariant into Multi-Cycle

- Step the single-cycle invariant (function) recursively
- Add a variable to keep track of the history of inputs
- Prove a lemma about what it means to compose a part of the function during the first cycle with another part of the function during the second cycle
- Leverage that lemma and the single-cycle invariant to enrich your invariant with a specification with what happens across two cycles
- Continue process until you've specified and proven as many cycles deep as you like



Example Code, 8-bit Multiplier

- Verilog
- Parsing the Verilog
- Definition: Step-n
- Lemma: One-cycle-invariant
- Theorem: Two-cycle-invariant-via-rewriting

Example Code, Verilog

```
module mul (result_ff, a, b, en, clk);

  output wire [15:0] result_ff;
  input [7:0] a, b;
  input en;
  input clk;

  wire [7:0] a_ff, b_ff;
  wire [15:0] result;
  wire [9:0] pp0;
  wire [11:2] pp1;
  wire [13:4] pp2;
  wire [15:6] pp3;
```

```
theflop #(8) a_flop (a_ff, a, clk);
theflop #(8) b_flop (b_ff, b, clk);

pp_mul pp0_mul (pp0, a_ff, b_ff[1:0], clk);
pp_mul pp1_mul (pp1, a_ff, b_ff[3:2], clk);
pp_mul pp2_mul (pp2, a_ff, b_ff[5:4], clk);
pp_mul pp3_mul (pp3, a_ff, b_ff[7:6], clk);

assign result = {6'b0,pp0} + {4'b0,pp1,2'b0}
              + {2'b0,pp2,4'b0} + {pp3,6'b0};

theflop #(16) result_flop (result_ff, result,
              clk);
```

```
endmodule
```

Example Code, Parsing the Verilog

```
(defsvtv mul-direct
    :design *mul*
    :inputs '(("clk"  0   ~)
              ("a"    a   _)
              ("b"    b   _))
    :overrides '(("pp0_mul.pp_in" pp0)
                 ("pp1_mul.pp_in" pp1)
                 ("pp2_mul.pp_in" pp2)
                 ("pp3_mul.pp_in" pp3)
                 ("result" o))
    :internals '(("pp0_mul.pp_in" _ pp0)
                 ("pp1_mul.pp_in" _ pp1)
                 ("pp2_mul.pp_in" _ pp2)
                 ("pp3_mul.pp_in" _ pp3))
    :outputs '(("result" _ o)))
```

Example Code, Definition: Step-n

```
(define step-n ((input-list input-list-p)
                (input-history input-list-p)
                (output-history output-list-p)
                (st-history st-list-p)
                (n natp))
  :returns (mv (input-history input-list-p :hyp :guard)
               (output-history output-list-p
                               :hyp :guard)
               (st-history st-list-p :hyp :guard))
  (cond ((zp n)
         (mv input-history output-history st-history))
        ((atom input-list)
         (mv input-history output-history st-history))
        ((atom st-history)
         (mv input-history output-history st-history))
        (t
         (mv-let
           (output new-st)
           (single-step (car input-list)
                        (car st-history))
           (step-n (cdr input-list)
                   (cons (car input-list)
                         input-history)
                   (cons output
                         output-history)
                   (cons new-st st-history)
                   (1- n))))))
```

Example Code, Lemma: One-cycle-invariant

- Proved automatically with GL and created corollary

```
(defthm one-cycle-invariant-corollary
  (implies (and (unsigned-8-p a)
                (unsigned-8-p b)
                (st-p st))
           (b* ((input (make-input :a a
                                   :b b))
                ((output new-st)
                 (single-step input st)))
             (and (equal
                   (st->pp0 new-st)
                   (* a (part-select b :low 0 :high 1)))
                  (equal
                   (st->pp1 new-st)
                   (* a (part-select b :low 2 :high 3)))
                  (equal
                   (st->pp2 new-st)
                   (* a (part-select b :low 4 :high 5)))
                  (equal
                   (st->pp3 new-st)
                   (* a (part-select b :low 6 :high 7)))
                  (equal
                   (output->o output)
                   (loghead 16
                            (+ (st->pp0 st)
                               (ash (st->pp1 st) 2)
                               (ash (st->pp2 st) 4)
                               (ash (st->pp3 st) 6)))))))))
```

Example Code, Theorem: Two-cycle-invariant-via-rewriting

```
(defruled two-cycle-invariant-via-rewriting
  (implies (and (unsigned-8-p a)
                (unsigned-8-p b)
                (unsigned-8-p nonsense-8-p-0)
                (unsigned-8-p nonsense-8-p-1)
                (st-p st))
           (b* ((input0 (make-input :a a
                                    :b b))
                (input1 (make-input :a nonsense-8-p-0
                                    :b nonsense-8-p-1))
                ((input-history output-history ?st)
                 (step-n (list input0 input1)
                         nil
                         nil
                         (list st)
                         2)))
             (equal (output->o (car output-history))
                    (* (input->a (cadr input-history))
                       (input->b (cadr input-history)))))))
```

```
:expand ((step-n (list (input a b)
                       (input nonsense-8-p-0 nonsense-8-p-1))
                 nil nil (list st)
                 2)
         (step-n (list (input nonsense-8-p-0 nonsense-8-p-1))
                 (list (input a b))
                 (list (mv-nth 0 (single-step (input a b) st)))
                 (list (mv-nth 1 (single-step (input a b) st))
                       st)
                 1))
:enable (an-arithmetic-property one-cycle-invariant-corollary))
```

Conclusion

- Symbolic Trajectory Evaluation (defsvtv) is great for data path verification
- Single-cycle invariants with recursive calls great for starvation proofs
  - Due in part to the power of ACL2 + GL + SAT
- Multi-cycle invariants perhaps necessary* for reasoning about a mixture of control logic and datapath

* Recent FSM additions to svtv library, GLMC, etc. may subsume this need

Hardware and Software
Engineered to Work Together

Backup

Abstract

Since the verification of the division/square-root unit, Oracle has focused on verifying control-oriented logic. We will describe how we used the tools stack and hand-written model for parts of the circuit to verify one of our control logic units. Taking this approach a step further, we hope to illustrate with a simple example how one can extract a single-cycle invariant and then grow that invariant to describe a circuit's behavior across multiple clock cycles.

ORACLE