A Flexible Formal Verification Framework for Industrial Scale Validation

Anna Slobodová
July 12, 2011
Centaur Technology, Inc.
anna@centtech.com

Joint work with
Jared Davis, Warren Hunt and Sol Swords

Outline

About Centaur Technology, Inc.

- Based in Austin, TX, USA
- Owned by Via Technologies, Inc.
- X86 Microprocessor Design
  implemented by AMD, Intel and VIA only
- About 100 engineers specify, design validate, bring up, test, build burn-in fixtures – everything but manufacturing
  - RTL logic team 20
  - Validation team 20
  - Transistor-level design team 25
  - Formal verification team 3

  and tens of contractors

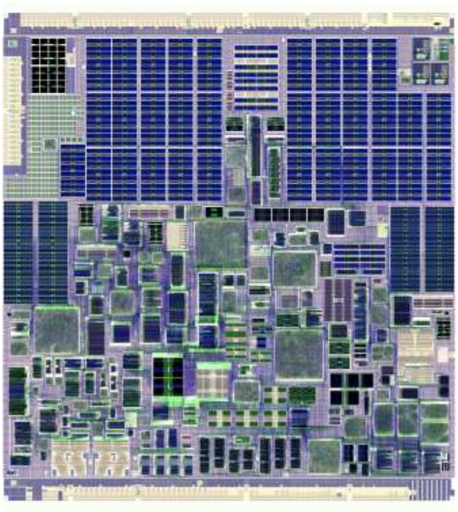
VIA Isaiah – X86-64 Microprocessor

X86 designs are complicated

- Intel 64-compatible
  I am not aware of existence of any formal X86 specification, despite several attempts to write one
- Intel VMX-compatible design
- Latest SSEx instructions
- Complex micro-architecture for performance
- Microcode
- Low cost, small size, low power, AND high performance – require custom design

Targeted at low-power, low-cost products:
netbooks, low-power workstations, and embedded designs.

VIA Nano^{TM} Microprocessor



Contemporary Example

- Full X86-64 compatible two-core design
- 40nm technology, 97.6 million transistors per core (195.7)
- AES, DES, SHA, and random-number generator hardware
- Built-in security processor
- Runs 40 operating systems, four VMs

Status of FV of Microprocessor Design (bird's eye view)

- **IBM**:
  - *Sixth Sense* – very sophisticated equivalence- and model-checking technology, with a limited use of theorem proving
  - Protocol verification using Murphi
- **AMD**: ACL2 based verification in a narrow area of FP arithmetics
- **Intel**: Probably the heaviest use of formal methods in industry
  - Sequential Equivalence-checking deployed everywhere
  - Model-checking developed by researchers and used by FV experts and by designers in ASIC teams
  - Protocol verification using Murphi and TLC
  - Microcode verification

Different Business Models of FV

- **IBM**: Mostly their own FV tools developed by big teams
  Projects set requirements for passing design through FV
- **AMD**: Small team of highly skilled researchers; use ACL2
  Not much deviation from their original focus on arithmetics
- **Intel**: Huge investment into big highly trained teams and growing
  - Own CAD tool company that provides all FV tools
  - Research ⇒ Development ⇒ Project CAD teams
  - Center of FV expertise with cross-project reach
  - Local FV experts

Who can afford formal methods?

- People with formal verification training are costly
- Building own FV tools is expensive and requires years of investment
- FV tools from CAD vendors
    - expensive
    - limited on-site support
    - often need tailoring to in-house design methodology
    - one still needs FV experts to run them

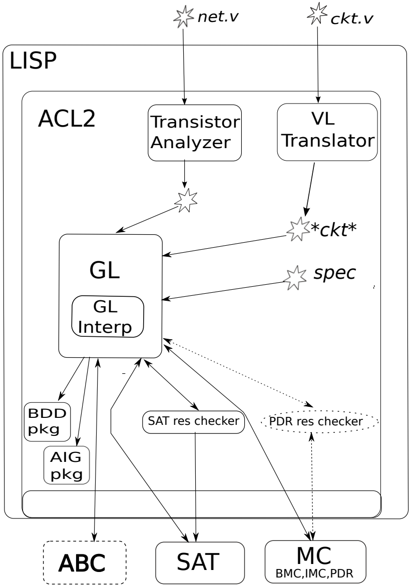
Who can afford formal methods?

- IBM, Intel,...
- Centaur Technology...
- You can afford it too

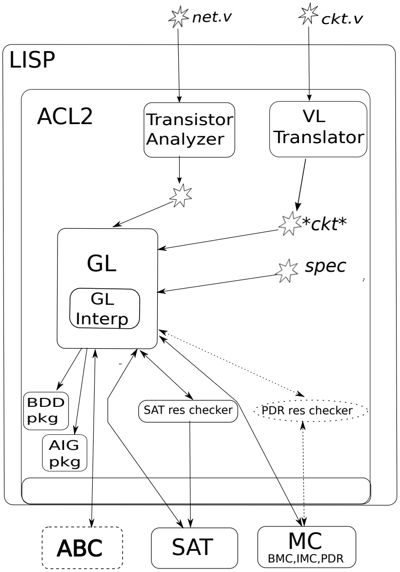
It is all about the business model!

- Use **extensible open source** tools
- Hire enthusiastic FV experts
- Point to the right problems

FV Framework

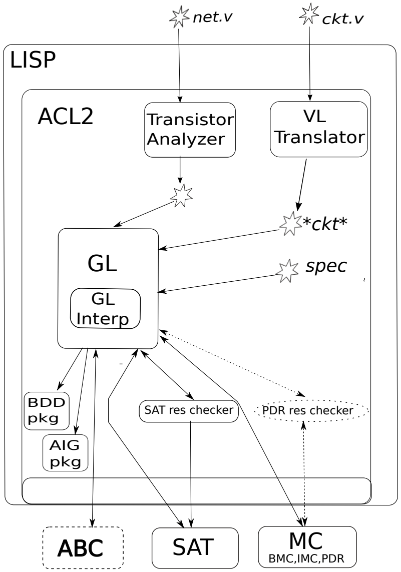


ACL2



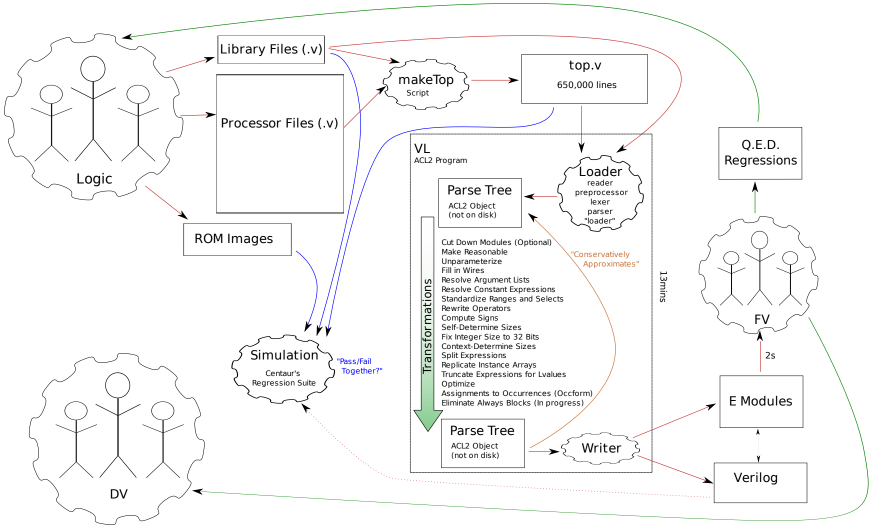
- Programming language
  - subset of LISP (CCL)
  - executability
  - reflection
- 1st order logic
- Theorem prover support (Austin)
- 100 man/year effort
- hardened in industrial environment (AMD, Rockwell-Collins, Centaur)

VL tool kit

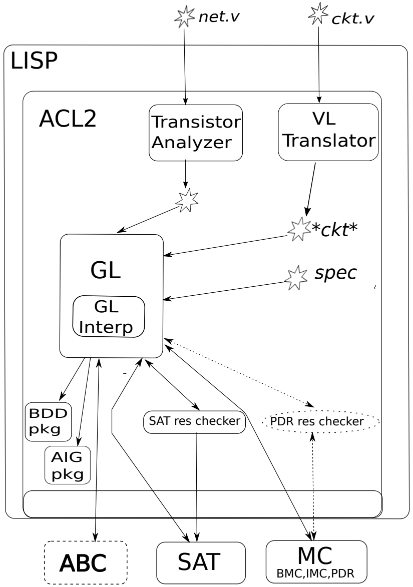


- While not formal, many theorems about translation
- Synthesis like aproach without optimization
- 650,000 lines of Verilog code
- Creates an ACL2 constant with semantics given by E interpreter
- Translation: 13 minutes
- Loading: couple of seconds
- Linting tool on top of translator

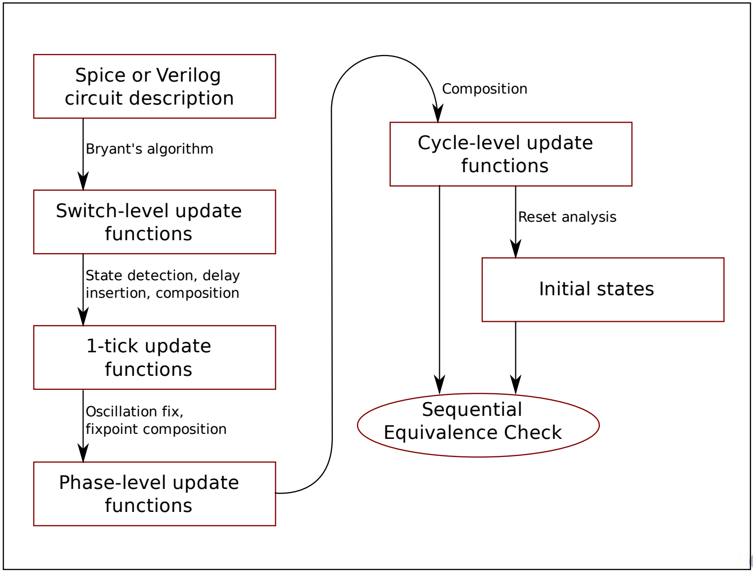
Verilog-to-E Translator



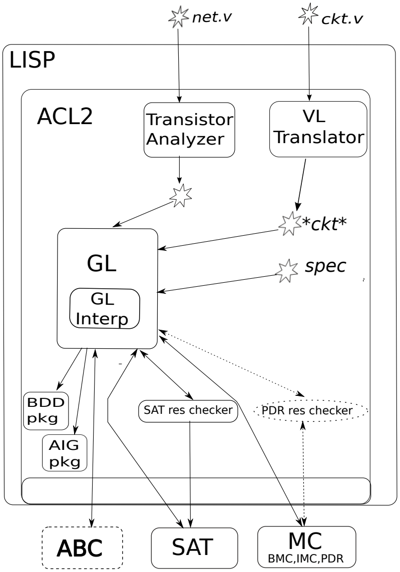
Transistor Analyzer



Transistor Analyzer


Spice or Verilog circuit description
Bryant's algorithm
Switch-level update functions
State detection, delay insertion, composition
1-tick update functions
Oscillation fix, fixpoint composition
Phase-level update functions
Composition
Cycle-level update functions
Reset analysis
Initial states
Sequential Equivalence Check

GL System



- Symbolic execution framework for proving theorems over objects from a finite domain
- Verified clause processor – creates an ACL2 theorem
- Automates discharge of low-level properties
  - makes proofs robust to design changes
  - requires little understanding of the design details
  - counterexample if fails

Example: Counting Bits
S. Anderson: Bit Twiddling Hacks

```
v = v - ((v >> 1) & 0x55555555);
v = (v & 0x33333333) + ((v >> 2) & 0x33333333);
c = ((v + (v >> 4) & 0xF0F0F0F) * 0x1010101) >> 24;
```

```
(defun fast-logcount-32 (v)
  (let* ((v (- v (logand (ash v -1) #x55555555)))
         (v (+ (logand v #x33333333)
               (logand (ash v -2) #x33333333))))
    (ash (32* (logand (+ v (ash v -4)) #xF0F0F0F)
              #x1010101)
         -24)))
```

```
(defun 32* (x y)
  (logand (* x y) (1- (expt 2 32))))
```

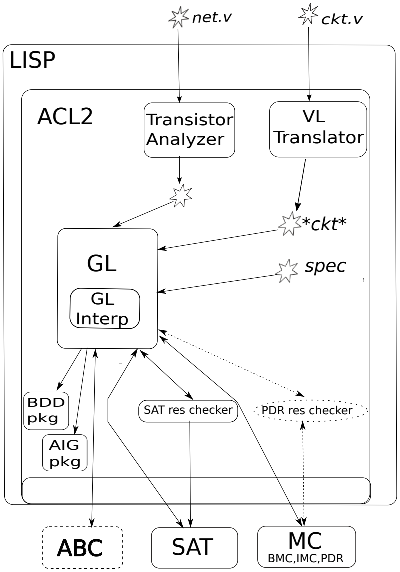
Example: continued

```
(def-gl-thm fast-logcount-32-correct
  :hyp (unsigned-byte-p 32 x)
  :concl (equal (fast-logcount-32 x)
                (logcount x))
  :g-bindings `((x ,(g-int 0 1 33))))
```

The proof completes in 0.09 seconds and results in the ACL2 theorem:

```
(defthm fast-logcount-32-correct
  (implies (unsigned-byte-p 32 x)
           (equal (fast-logcount-32 x)
                  (logcount x)))
  :hints ((gl-hint ...)))
```

GL System



- Returns an ACL theorem or a counterexample
- Various features: case splitting, parametrization
- Offers a choice between BDD and SAT solution
  - verified BDD package
  - SAT with verified result
  - SAT without guarantee

Binary Decision Diagram and And-Inverter Graph packages

- operations proven correct w.r.t. BDD and AIG evaluation

$$\forall x \in B^n : (f \otimes g)(x) = f(x) \times g(x)$$

  f and g are BDDs/AIGs;
  $\otimes$ is a Boolean operation over BDDs/AIGs;
  $\times$ the respective Boolean operation.

- performance
  - hash-consing
  - memoization
  - lisp garbage collection

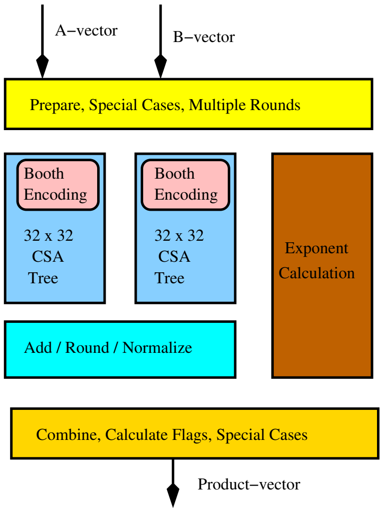
Examples of Problems

- Verification of Arithmetic Circuits
- RTL-to-RTL Equivalence Checker
- Late Changes in the Design
- Clock Tree Analysis

Verification of Arithmetic Circuits

- All proofs use strength of ACL2 with design with GL System - either BDD or SAT, used to discharge “low”-level properties
- Complexity of the design
  - High-level algorithm structure often lost in low-level optimizations
  - Brute-force extraction of equations does not work
  - Design is not stable - changing while proofs are developed
- Clarifying specification - X86 instructions are not the same as micro-operations
- Most of arithmetic, logic and misc micro-operations verified
  - FADD/FSUB verification
  - Verification of Integer and Floating-Point Multipliers
  - Verification of MMX and IU
- Proofs run at least once a week
- Proofs highly portable to future generation designs

Verification of High-Performance Multipliers
Complexity - inherent in function and in design



- Multiplication function is beyond the capacity of BDDs and SAT-solver
- Requires decomposition
- Boundaries not clear, sometimes spread over time
- No automatic way of finding properties on the decomposition boundary
- Requires the proof of the multiplication algorithm
- Pipelined design might cause a reconfiguration of the multiplier every cycle

Verification of Multipliers (continued)

Several Multipliers, many multiplier configurations for variety of pipelined operations

- signed and unsigned integer multiply: up to 64x64
- packed-integer multiply
- packed-integer multiply-and-add
- floating-point: X87 and SSEx flavors with single, double, and extended precisions

All verified using GL-System with BDDs

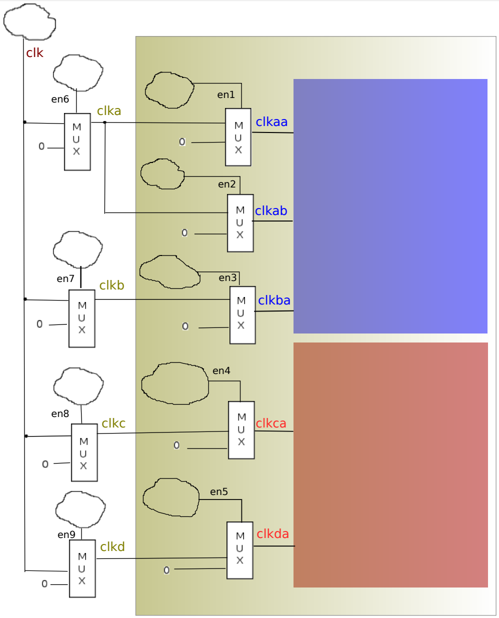
RTL-to-RTL Equivalence checker

- Motivation:
  - Changes in RTL design reflect our everyday reality – fixing functional bugs, fixing timing, aid to equivalence-checker
  - Often within latch boundaries
  - Riskier in later stages of the design
- Solution: RTL-designer-friendly Combinational Equivalence Checker
  - First version was put together within couple of days
  - Then tuned for easy use - no FV knowledge required
  - Counterexamples feed Verilog simulator to ease debugging
- Extensible to sequential equivalence checker

Late Changes in the Design

- Problem: Bug escapes always happen. The later the more costly!
- Bug fixes
  - In microcode
  - Changing transistors – changing design masks VERY COSTLY!
  - Spare transistors/gates in the design to be used for late changes.
- Can we help with the last solution? Automate the slow tedious process done by senior designers.
  - Given: an RTL, gate-network implementation and changes in the RTL
  - Goal: find equations consisting of the network gates that implement the RTL change
- Solution: using our equivalence-checking capabilities, we find mappings from RTL signals to network gates, or an equation containing the gates
  Typically runs in minutes.

Clock Tree Analysis



Summary

- ACL2 based FV framework used at Centaur
  - **Flexibility** to implement different tools and prove their correctness
  - **VL-Translator** builds a formal model of the RTL design
  - **Transistor Analyzer** builds a formal model from the transisto-level design
  - **GL-system** equipped with BDD pkg and SAT solver
  - **Correctness of arithmetic circuits**
  - Various **problem-driven tools** have been developed
  - **External tools** are used were we need more capabilities
- Future – driven by company's needs
  - Extend proofs to other areas
  - Make our tools more robust and user friendly
  - Gain more influence on design methodology

Conclusion

- FV can be done in a small/medium size company
- Choice of framework/tools/language is crucial
- Extensibility – most important
- Recognition that FV cannot solve all problems (yet). Choose those with high return first.
- Re-use, strengthen, extend, automate
- Keep pushing the boundary

Acknowledgement

We wish to acknowledge:
Bob Boyer, Gary Byers, Niklas Een, Matt Kaufmann, Alan Mishchenko