Relational STE and Theorem Proving for Formal Verification of Industrial Circuit Designs

John O’Leary and Roope Kaivola, Intel
Tom Melham, Oxford
CPU datapath verification at Intel

• Thousands of operations
  – Integer, FP, SSE, AVX, ...
  – “Miscellaneous”
  – Various operating modes, flags, faults

• Live RTL, changing frequently until a few weeks before tapeout
Scaling up

- Tens of designs
- Different optimization points
- Different teams
- Different countries
- Not only CPUs
- Not all have FV experts on staff
Integer multiplier

\[ S_1 = \sum BE_i \times 2^{ki} \]

\[ P = \sum PP_i \times 2^{ki} \]

\[ PP_i = S_2 \times BE_i \]
The multiplier zoo

- 10-20 multipliers
- Hand designed
- Hand optimized
- All different
FV challenges

• Varying specs and verification strategies
  – Implementation changes from design to design
  – Multiplier always requires decomposition

• Ten designers but not ten multiplier FV experts

• Same story for integer, MMX, FP, SSE, GPU flavors of multiplication, addition, division, ...
  – Some operations require even more intricate decomposition
The solution

Parameters \rightarrow CVE \rightarrow \text{Per-design specs Verification runs} \rightarrow \text{Deduction} \rightarrow \text{specs+runs} \Rightarrow \text{correct}

Development \rightarrow \text{Regression}
The solution *done right*

- An executable logic for writing the specs and verification scripts: reFLect
- A symbolic simulator that admits relational specifications written in logic: rSTE
- A tightly integrated theorem prover for executing the deductive proofs: Goaled
The solution *done right*

- An executable logic for writing the specs and verification scripts: *applicative common lisp*
- A symbolic simulator that admits relational specifications written in logic: *ESIM+GL*
- A tightly integrated theorem prover for executing the deductive proofs: *ACL2*

[Slobodová et al, MEMOCODE’11]
The reFLect Language

• Core syntax:

\[ n,o,p ::= k \mid v \mid n\ o \mid \lambda p.\ n\ o\ |\ \langle n \rangle \mid ^{n:}\sigma \]

pattern matching reflection

• ... plus extensions driven by necessity
  – BDDs built in as a primitive type
  – Quotient types
  – Overloading
  – Named function parameters
  – Records
  – Possibly unsafe features: references, I/O, recursion
Higher Order Logic of reFLect

• HOL, following Church:

\[
\text{Logic} = \begin{cases} 
\lambda\text{-calculus} \\
\quad + \ 	ext{logical constants} \\
\quad + \ 	ext{rules}
\end{cases}
\]

• The reFLect logic:

\[
\text{Logic} = \begin{cases} 
\text{reFLect} \\
\quad + \ 	ext{logical constants} \\
\quad + \ 	ext{rules}
\end{cases}
\]

• Basic idea in both systems:

\[ n \to p \] means \[ \vdash n = p \]
Define \( \forall, \exists \), etc by axioms
Add rules for function equality

Proof by evaluation
Goaled Theorem Prover

• LCF-style implementation, following in the footsteps of HOL and HOL Light
  – Thm is a protected data type, constructible only through a small set of trusted function calls (a.k.a. inference rules)

• Features driven by necessity
  – Theories: of reFLect data types, natural numbers, integers, rationals, lists, pairs, reFLect ADTs
  – Proof automation: rewriting, first order solving, linear arithmetic
  – Bitstring arithmetic
  – Support for the reflect language extensions
The last bit

• An executable logic for writing the specs and verification scripts: reFLect

• A symbolic simulator that admits relational specifications written in logic: rSTE

• A tightly integrated theorem prover for executing the deductive proofs: Goaled
Limitations of STE

- Trajectory assertion:
  - $\text{ckt} \models [\lbrack S \text{ is } v \implies (BE_i \text{ is } f_i(v)) \rbrack]$

- But,
  - You need a special purpose reasoning system for this special purpose logic
  - Relational specifications cannot be expressed directly

$$S = \sum_{i=0}^{N-1} BE_i \times 2^{ki}$$
Relational STE

• STE’s antecedent and consequent are replaced with lists of constraints
  – A constraint is a relationship between a finite set of circuit nodes at specified points in time
• Idea:
  – $rSTE \text{ ckt cin cout}$ means “In any behavior of $ckt$ in which all of the constraints $cin$ hold, all of the constraints $cout$ hold”
Relational STE Intuition

\[
\begin{align*}
&(ci, 1) \\
&(a, 1) \\
&(b, 1) \\
&(s, 2) \\
&(c, 2)
\end{align*}
\]

rSTE ckt

\[
\begin{align*}
\text{["! (ci, 1)"] } \quad &\text{["(a, 1) + (b, 1) = (s, 2) + 2 \times (c, 2)"]}
\end{align*}
\]
Constraints

• A constraint $c$ has three components:
  – $\text{name}(c) : \text{string}$
  – $\text{sig}(c) : (\text{string} \times \text{num}) \text{ list}$
  – $\text{pred}(c) : ((\text{string} \times \text{num}) \to \text{bool}) \to \text{bool}$

• The behavior of the circuit is also formulated as a constraint:
  $\llbracket \text{ckt} \rrbracket : ((\text{string} \times \text{num}) \to \text{bool}) \to \text{bool}$
From Relational STE to Logic

• Theorem:

\[
\forall ckt \ cin \ cout.
\]

\[
rSTE \ ckt \ cin \ cout \ \Rightarrow
\]

\[
\forall e. \ [ckt]e \ \Rightarrow
\]

\[
\text{predl} \ cin \ e \ \Rightarrow \ \text{predl} \ cout \ c
\]

• For lists of constraints,
  
  – predl [] e \ \triangleq \ T
  
  – predl (c::cs) e \ \triangleq \ \text{pred}(c) \ e \ \land \ \text{predl}(cs) \ e
Relational STE in Action

• Define $boothc$ such that
  \[ \text{pred}(boothc) = \lambda e. \text{eqn1}(s2i \ e \ s1) \]
  \[ \text{eqn1}(x) \triangleq (x = \sum_{i=0}^{N-1} BE_i(x) \times 2^{ki}) \]

• Then, $rSTE \ ckt[] \ [boothc] \rightarrow T$
  implies
  \[ \forall e. [ckt]e \Rightarrow \text{predl} [] e \Rightarrow \text{predl} [boothc] e \]
Relational STE in Action

- $\forall e. \text{ckt} e \Rightarrow \text{predl} [] e \Rightarrow \text{predl} [\text{boothc}] e$
- $\forall e. \text{ckt} e \Rightarrow \text{pred} (\text{boothc}) e$
- $\forall e. \text{ckt} e \Rightarrow \text{eqn1}(s2i e s1)$
- $\forall e. \text{ckt} e \Rightarrow$
  $$s2i e s1 = \sum_{i=0}^{N-1} BE_i(s2i e s1) \times 2^{ki}$$
Completing a Multiplier proof

∀ \( e \). \([ckt]\)e \Rightarrow
(\bigwedge s2i \ e \ pp_i = BE_i(s2i \ e \ s1) \times s2i \ e \ s2)

∀ \( e \). \([ckt]\)e \Rightarrow
(s2i \ e \ prod = \sum_{i=0}^{N-1}(s2i \ e \ pp_i) \times 2^{ki})

∀ \( e \). \([ckt]\)e \Rightarrow
(s2i \ e \ s1 = \sum_{i=0}^{N-1} BE_i(s2i \ e \ s1) \times 2^{ki})

∀ \( e \). \([ckt]\)e \Rightarrow
(s2i \ e \ prod = s2i \ e \ s1 \times s2i \ e \ s2)
Proof engineering

• Additional arguments to rSTE
  – Constant antecedent: clock, reset
  – rSTE options: bdd variable ordering, param, ...
  – Not shown here, but see paper

• Analysis of CVE verification scripts
  – N layers of function calls between input parameters and generation of specs
  – Much deductive effort toward exposing the specs
  – Routine rewriting, also not shown here
Status and prospects

• reFLect and rSTE are the main workhorses of datapath verification across Intel
• Frameworks for integer and FP multipliers, FMAs, adders, divide/sqrt are widely deployed
• Goaled checking of integer multipliers is used on a mainline design project and being pushed to others
• We plan to integrate Goaled checking with our other frameworks