HIGH ASSURANCE HARDWARE WITH REWIRE

Just Say No! to Semantic Archaeology

Bill Harrison, Adam Procter, Ian Graves, & Michela Becchi
University of Missouri

Gerard Allwein
US Naval Research Laboratory
Archaeology
Semantic Archaeology & Formal Methods

System Implementation

Mathematical System Model formulating precise system security properties
“The key difficulty was to go from the informal-prose vendor documentation, with its often-tantalising ambiguity, to a fully rigorous definition (mechanised in HOL) that one can be reasonably confident is an accurate reflection of the vendor architectures (Intel 64 and IA-32, and AMD64).”

Semantic Archaeology as It Occurs in Nature

x86 Instruction Semantics in HOL in terms of Monadic Microcode; e.g.,

seq\text{T} : 'a M → ('a → 'b M) → 'b M
par\text{T} : 'a M → 'b M → ('a * 'b)M
const\text{T} : 'a → 'a M
failure\text{T} : unit M
map\text{T} : ('a → 'b) → 'a M → 'bM
lock\text{T} : unit M → unit M
Security Flows in the Many Core Era*

- Highly (Re)configurable Architectures/FPGAs
- Many Specially Tailored, “One Off” Components
  - Reuse of Off-the-shelf components
  - “Mix and Match” comes to Hardware
- Challenge: High Assurance in this environment
  - Want the flexibility and speed of development
  - …but also want formal guarantees of security and safety for critical systems.

* Funded by the Office of the Assistant Secretary of Defense for Research and Engineering
Hardware Synthesis from Domain Specific Languages

- **Delite** [Olukotun, Ienne, et al.]
  - DSLs and Language Virtualization
  - “The Three P’s”

- **ReWire**
  - Fourth P: *Provability*
  - DSL with rigorous semantics
    - Modular Monadic Semantics
  - High assurance
    - Security & safety properties
    - Formal methods Productivity
ReWire Language & Toolchain

- Inherits Haskell’s good qualities
  - Pure functions, strong types, monads, equational reasoning, etc.
  - Formal denotational semantics [HarrisonKieburtz05,Harrison05]
- Language design identifies HW representable programs
  - Mainly restrictions on recursion in functions and data
  - Built-in types for HW abstractions incl. clocked/parallel computations
Xilinx PicoBlaze Architecture

### PicoBlaze Data Layout in ReWire

```haskell
import Data.Record

type RegFile = Table W4 W8

import Data.Sequence as Seq

data Stack = Stack { contents :: Table W5 W10, pos :: W5 }

data Inputs = Inputs { instruction_in :: W18, in_port_in :: W8, interrupt_in :: Bit, reset_in :: Bit }

data Outputs = Outputs { address_out :: W10, port_id_out :: W8, write_strobe_out :: Bit, out_port_out :: W8, read_strobe_out :: Bit, interrupt_ack_out :: Bit }
```

Expressing Architectural Designs in ReWire

Details in “Semantics-directed Architecture in ReWire”, Procter et al., ICFPT13
Expressing Architectural Designs in ReWire (cont’d)

Details in “Semantics-directed Architecture in ReWire”, Procter et al., ICFPT13

- `fde` device is tail-recursive
- Clock timing is expressed in `Dev` monad

Xilinx PicoBlaze Architecture

```
fde :: Dev Inputs PicoState Outputs
fde = do s <- getPicoState
    let i = inputs s
        instr = instruction_in i
        ie <- getFlagIE
        if reset_in i == 1
            then reset_event
            else if ie == 1 &&
                interrupt_in i == 1
                    then interrupt_event
                else decode instr
    fde
```

PicoBlaze Fetch-Decode-Execute in ReWire
Compare with PicoBlaze in VHDL

- Outermost VHDL Component for PicoBlaze

```vhdl
component KCPSM3
port (  
    instruction : in std_logic_vector(17 downto 0); -- Inputs type  
    in_port : in std_logic_vector( 7 downto 0);  
    interrupt : in std_logic;  
    reset : in std_logic;  
    clk : in std_logic;  
    address : out std_logic_vector( 9 downto 0); -- Outputs type  
    port_id : out std_logic_vector( 7 downto 0);  
    write_strobe : out std_logic;  
    out_port : out std_logic_vector( 7 downto 0);  
    read_strobe : out std_logic;  
    interrupt_ack : out std_logic;
);
end component;
```

- Corresponds to ReWire term of monadic type
  - **Dev Inputs PicoState Outputs**

**Crucial Distinction:**
Dev is a formal object we can reason about.
Types for Devices

Terms of type $(\text{Dev } i \ s \ o)$ compiled to...

```
newtype ReT i o m a = ReT (m (Either a (o, i \rightarrow ReT i o m a)))
newtype StT s m a = ...

type Dev i s o = ReT i o o \ (\text{StT } s \ \text{Identity}) ()
```
Performance

• Prototype ReWire compiler vs. Hand-coded VHDL implementation by experienced Xilinx engineer.
  • XST synthesis tool for Spartan-3E XC3S500E, speed -4
  • configured to optimize for speed, not space.

<table>
<thead>
<tr>
<th></th>
<th>Slices</th>
<th>Flip Flops</th>
<th>4-LUTs</th>
<th>$F_{max}$ (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PicoBlaze</td>
<td>99</td>
<td>76</td>
<td>181</td>
<td>139.919</td>
</tr>
<tr>
<td>ReWire</td>
<td>451</td>
<td>110</td>
<td>866</td>
<td>69.956</td>
</tr>
</tbody>
</table>
Designing a Secure Dual-core PicoBlaze*

- Two PicoBlazes (L ≤ H) with a shared register Reg
  - Reg is read-only by H; read+write by L
- Proved a non-interference style security specification
  - Equational proof based on “by-construction” properties of monads
  - Verifies ReWire code directly
  - Just say NO! to Semantic Archaeology.

Designing a Secure Dual-core PicoBlaze*

- Type of Dual-Core constructor function:

\[
\text{dualcore} :: \text{Dev Inputs PicoState Outputs} \rightarrow \\
\text{Dev Inputs PicoState Outputs} \rightarrow \\
\text{Dev2 Inputs PicoState Outputs}
\]

Security Theorem

\[ \text{pull os is (dualcore lo } \text{hi}) \gg= \kappa_0 \]
\[ = \text{pull os is (dualcore lo } \text{nop}) \gg= \kappa_0 \]

where

\[ \kappa_0 = \lambda \text{os. mask}_H \gg \text{return os} \]
\[ \text{nop} = (\text{skip o}_0 \text{ i}_0) \]

Proof follows closely:
Proof Sketch of Security Theorem

\[
pull \ os \ [i_1, \ldots, i_n] \ (\text{dualcore} \ lo \ hi) \ >>= \ \lambda \ os. \ mask_H \ >> \ return \ os
\]
\[
= (lh_1; \ldots; lh_n) >>= \ \lambda \ os. \ mask_H \ >> \ return \ os
\]
\[
= (lh_1; \ldots; lh_n; mask_H) >>= \ \lambda \ os. \ mask_H \ >> \ return \ os
\]
\[
= (lh_1; \ldots; l_n; mask_H) >>= \ \lambda \ os. \ mask_H \ >> \ return \ os
\]
\[
= (lh_1; \ldots; mask_H; l_n) >>= \ \lambda \ os. \ mask_H \ >> \ return \ os
\]
\[
= (lh_1; mask_H; \ldots; l_n) >>= \ \lambda \ os. \ mask_H \ >> \ return \ os
\]
\[
= (l_1; mask_H; \ldots; l_n) >>= \ \lambda \ os. \ mask_H \ >> \ return \ os
\]
\[
= (l_1; \ldots; l_n) >>= \ \lambda \ os. \ mask_H \ >> \ return \ os
\]
\[
= pull \ os \ [i_1, \ldots, i_n] \ (\text{dualcore} \ lo \ \text{nop}) \quad >>= \quad \lambda \ os. \ mask_H \ >> \ return \ os
\]
Performance

- Comparing the single core PicoBlaze to the dual core:

<table>
<thead>
<tr>
<th></th>
<th>Slices</th>
<th>Flip Flops</th>
<th>4-LUTs</th>
<th>$F_{max}$ (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-Core</td>
<td>907</td>
<td>258</td>
<td>1735</td>
<td>67.867</td>
</tr>
<tr>
<td>1-Core</td>
<td>451</td>
<td>110</td>
<td>866</td>
<td>69.956</td>
</tr>
<tr>
<td>Ratio</td>
<td>2.011</td>
<td>2.345</td>
<td>2.003</td>
<td>0.970</td>
</tr>
</tbody>
</table>
Hardware vs. Program Verification

**Traditional HW Verification**

- HW Verification has been around for many, many years...
  - HOL (Cambridge), Boyer-Moore (Texas), Isabelle (Cambridge & Munich), BDD’s, etc., etc.

**Basic Recipe**
1. Start with circuit,
2. Produce formal model capturing its essence,
3. Encode in theorem prover logic & verify!

- How do you check the faithfulness of Step 2?
  - Does the model capture the artifact?
  - Can you prove that it is faithful?

**Program Verification**

- Say you have a programming language,
- **IF** you have:
  - a compositional semantics for the language, and
  - a trusted compiler,
- **THEN** you can:
  - verify programs
  - verify compiler’s semantic faithfulness, and
  - produce high assurance implementations.

- Canonical example: Hoare semantics for procedural languages.
- This is the approach ReWire takes.
Fast Regular Expression Matching Using FPGAs

- Deep Packet Inspection for detecting malware
- Use HW Parallelism to Represent Non-determinism
- Sidhu & Prasanna 2001
- Becchi & Crowley 20[07|08|09|10]
  - Handwritten regular expression compiler in C
  - State of the art performance
Regular EXpression HArdware Compiler-Compiler

 rexhacc :: (NFA a -> NFA a) -> RegEx a -> ReWire

compiler :: RegEx a -> ReWire
compiler = rexhacc opt
  where opt = (o_1 o ... o o_n)

RexHacc Performance Evaluation

Details in “Hardware Synthesis from Functional Embedded Domain-Specific Languages:
ReWire & Proof Engineering

- Proof Engineering
  - Rewire both...
    - Computational λ-calculus
  - Unifies specification, design & implementation languages

- ARM spec. [Fox/Myreen10,...]
  \[
  \text{arm instr} : \\
  \text{iid} \quad \rightarrow \\
  \text{encoding } \times \text{bool}[4] \times \text{instr} \rightarrow \\
  \text{unit M}
  \]

- Collaboration with Australian DSTO laboratory
THANKS!

Joint work with Dr. Gerry Allwein of US Naval Research Laboratory and Dr. Michela Becchi, Dr. Adam Procter, and Ian Graves of MU
Papers

- Distributed Logic. Allwein and Harrison. NRL Memo Report, 2014