Implementing DIFC with PCHIP
A Novel Proof Framework

Eric J. Love

Yale University

April 5, 2011
Motivation

- Previous formulation in Coq inadequate for real tasks
- Need to clean up proof framework
- Biggest mistake: don’t use Coq to “compute”!
- Build a new formal semantics model first
- Use Coq later for implementation
- Also, want to extend semantics for information flow
Start from scratch: Formalization of Types

- value = \{lo, hi\} : Type
- \(\forall_n \text{bits } n \triangleq \mathbb{N}_n \rightarrow \text{value} : Type\)
- \(\forall_n \text{signal } n \triangleq \mathbb{N} \rightarrow \text{bits } n : Type\)
- \(\forall_n \text{wire } n : Type\)
- \(\text{SEnv} \triangleq \forall_n \text{wire } n \rightarrow \text{signal } n : Type\)
- \(\text{module} \triangleq \text{SEnv} \rightarrow \text{SEnv} : Type\)
In order to discuss semantics of our Verilog formulation, we must first introduce the concept of a *typing judgement*.

**Definition**

A *typing judgement* of the form $H ⊢ s : T$ is a representation of the knowledge that signal $s$ has the type $T$ in environment $H$. By *environment*, we mean the inside of a module's main body.
We assume that a wire or reg can only be declared once within a module (must be checked by compiler, otherwise the following inference rules will be unsound).

\[
\begin{align*}
\langle \text{module } H (\ldots) C \rangle \\
\vdash H : \text{module}
\end{align*}
\]

\[
\begin{align*}
\langle \text{module } H (\ldots) \text{ begin } \ldots \times \text{wire}[(n+1):0];\ldots \text{ end} \rangle \\
H \vdash x : \text{wire } n
\end{align*}
\]
Type Inference Rules for Expressions

Within a module’s body, each expression has a type associated with it. Here are some basic inference rules.

\[
\begin{align*}
H & \vdash w : \text{wire } n \quad E = w \\
\hline
H & \vdash E : \text{wire } n
\end{align*}
\]

\[
\begin{align*}
H & \vdash E : \text{wire } n \\
\hline
H & \vdash \neg E : \text{wire } n
\end{align*}
\]

\[
\begin{align*}
H & \vdash E_1 : \text{wire } n \\
H & \vdash E_2 : \text{wire } m \\
\hline
H & \vdash E_1 \equiv E_2 : \text{wire } 1
\end{align*}
\]

\[
\begin{align*}
op & \in \{\&, \hat{}, |\}
\end{align*}
\]

\[
\begin{align*}
H & \vdash E_1 : \text{wire } n \\
H & \vdash E_2 : \text{wire } m \\
n & \leq m \\
\hline
H & \vdash E_1 \text{ op } E_2 : \text{wire } n
\end{align*}
\]
Define an operator $[H \triangleright E : \text{wire } n] : \text{SEnv} \rightarrow \text{signal } n$

- Returns a function mapping environments to signals
- Semantics of expressions parametrized on past history of signals in containing module
  (Represented by SEnv variable $\rho$)
Semantic Equivalence Relations

- Define an equivalence relation $\equiv^n$ between signals:
  \[
  s_1 \equiv^n s_2 \iff \forall t, k < n s_1(k, t) = s_2(k, t) \tag{1}
  \]
  (Recall that $s_1$ is a function $\mathbb{N}_n \to \mathbb{N} \to \text{value}$. The reflexive, symmetric, and transitive properties clearly hold for $\equiv^n$)

- Define another equivalence relation $\equiv$ on expression semantic functions:
  \[
  \llbracket H' \gg E' : \text{wire } n \rrbracket \equiv \llbracket H \gg E : \text{wire } n \rrbracket \iff
  \forall \rho \llbracket H' \gg E' : \text{wire } n \rrbracket(\rho) \equiv^n \llbracket H \gg E : \text{wire } n \rrbracket(\rho) \tag{2}
  \]
  Recall that $\vdash \llbracket H \gg E : \text{wire } n \rrbracket(\rho) : \text{signal } n$ and is thus a mapping of clock cycle ($\mathbb{N}$) onto a set of bits (bits $n$).
Some Useful Semantic Primitives

- Define $V_{dd}$ as an expression that always evaluates to $\text{hi}$:

  \[
  \llbracket H \triangleright V_{dd} : \text{wire 1} \rrbracket \rho t = \text{hi}
  \]

- Similarly, $G_{nd}$ always $\text{lo}$:

  \[
  \llbracket H \triangleright G_{nd} : \text{wire 1} \rrbracket \rho t = \text{lo}
  \]
Further Semantics: $\rho t$-Abstractions

- Want to define semantics w/o referencing $\rho$ and $t$ constantly
- A $\rho t$ abstraction is a semantic definition parametrized w.r.t. these two var’s and has type 
  $(\text{SEnv} \rightarrow \text{signal } n) \rightarrow (\text{SEnv} \rightarrow \text{signal } m) \rightarrow (\text{SEnv} \rightarrow \text{signal } l)$
  for some $n$, $m$, and $l$ in the case of a binary operator.
- Will allow us to map two $[ H \triangleright E : \text{wire } n ]$’s onto one and recursively define semantics of expression trees
- Also have abstractions of type 
  $(\text{SEnv} \rightarrow \text{signal } n) \rightarrow (\text{SEnv} \rightarrow \text{signal } m)$ for unary operators, such as NOT
Some $\rho t$-Abstractions for Selected Operators

- Give definitions for AND, OR, and NOT w/ semantic inference rules
- Definition of AND operator:

\[
F_1 = [H \triangleright E_1 : \text{wire } n_1] \quad F_2 = [H \triangleright E_2 : \text{wire } n_2] \\
F_1 \rho t k = \text{lo} \lor F_2 \rho t k = \text{lo} \\
\text{and}(F_1, F_2) \rho t k = \text{lo}
\]

\[
F_1 = [H \triangleright E_1 : \text{wire } n_1] \quad F_2 = [H \triangleright E_2 : \text{wire } n_2] \\
F_1 \rho t k = \text{hi} \land F_2 \rho t k = \text{hi} \\
\text{and}(F_1, F_2) \rho t k = \text{hi}
\]

- Imagine analogous definitions for OR and NOT...
Using Abstractions for Expression Semantics

- Consider the case of \([H \triangleright E_1 \& E_2 : \text{wire } n]\).
- Semantics described with abstraction as:

  \[
  \left[ H \triangleright E_1 \& E_2 : \text{wire } n \right] = \text{and} \left( \left[ H \triangleright E_1 : \text{wire } n_1 \right], \left[ H \triangleright E_2 : \text{wire } n_2 \right] \right)
  \]

- Other cases analogous:

  \[
  \left[ H \triangleright E_1 | E_2 : \text{wire } n \right] = \text{or} \left( \left[ H \triangleright E_1 : \text{wire } n_1 \right], \left[ H \triangleright E_2 : \text{wire } n_2 \right] \right)
  \]

  \[
  \left[ H \triangleright \neg E : \text{wire } n \right] = \text{not} \left( \left[ H \triangleright E : \text{wire } n \right] \right)
  \]
Representing Wire Assignment Statements

- For simple assignment statements we have a proof rule:

\[
\begin{array}{c}
\langle \text{module } H(\ldots) \ldots \text{ assign } w = E; \ldots \rangle \\
H \vdash w : \text{wire } n \quad H \vdash E : \text{wire } n \\
\hline
[H \triangleright w : \text{wire } n] \equiv [H \triangleright E : \text{wire } n]
\end{array}
\]

- We also introduce new rules for expressions with conditionals:

\[
E = \langle E_1 \ ? \ E_2 : E_3 \rangle \\
[H \triangleright E_1 : \text{wire } n_1] \rho t 0 = \text{hi} \\
H \vdash E_2 : \text{wire } n \quad H \vdash E_3 : \text{wire } n \\
\forall k < n [H \triangleright E : \text{wire } n] \rho t k = [H \triangleright E_2 : \text{wire } n] \rho t k
\]

- (Analogous rule for \text{lo} case of \text{E}_1 \ldots)
Non-Blocking Assignment: The Update Rule

Need to represent `always @ (...) begin ... end` blocks in proof rules

First, establish that command $C$ updates some register:

$$C = \langle w <= E \rangle$$
$$\text{upd}_C \left( w \leftarrow V_{dd} \ E \right)$$

Says that whenever $V_{dd}$ is hi (i.e. always), the command $C$ will update the value of $w$ to $E$.

Then define “no-update” condition for a command:

$$C = \langle w <= E \rangle \quad w' \neq w$$
$$\overline{\text{upd}_C \left( w' \leftarrow V_{dd} \right)}$$
**Sequential Statement Update Rules**

\[
C = \langle C_1; C_2 \rangle \quad \text{upd}_{C_2} \left( w \overset{E_1}{\leftarrow} \right) \quad \text{upd}_{C_1} \left( w \overset{E_1}{\leftarrow} E_2 \right)
\]

\[
\text{upd}_C \left( w \overset{E_1}{\leftarrow} E_2 \right)
\]

\[
C = \langle C_1; C_2 \rangle \quad \text{upd}_{C_2} \left( w \overset{E_2}{\leftarrow} \right) \quad \text{upd}_{C_1} \left( w \overset{E_1}{\leftarrow} E_2 \right)
\]

\[
\text{upd}_C \left( w \overset{E_1}{\leftarrow} E_2 \right)
\]

\[
C = \langle C_1; C_2 \rangle \quad \text{upd}_{C_1} \left( w \overset{E}{\leftarrow} \right) \quad \text{upd}_{C_2} \left( w \overset{E}{\leftarrow} \right)
\]

\[
\text{upd}_C \left( w \overset{E}{\leftarrow} \right)
\]
If-Statement Update Rules

\[
C = \langle \text{if } (B) \ C_1 \text{ else } C_2 \rangle \quad \text{upd}_{C_1} \left( w \leftarrow^{E_1} E_2 \right) \\
\text{upd}_C \left( w \leftarrow^{B \& E_1} E_2 \right)
\]

\[
C = \langle \text{if } (B) \ C_1 \text{ else } C_2 \rangle \quad \text{upd}_{C_2} \left( w \leftarrow^{E_1} E_2 \right) \\
\text{upd}_C \left( w \leftarrow^{(\neg B) \& E_1} E_2 \right)
\]
Module Update Rules

\[ \langle \text{module } H(\ldots) \ C \rangle \quad \text{upd}_C \left( w \leftarrow E_1 \ E_2 \right) \]

\[ H \triangleright w \leftarrow E_1 \ E_2 \]

\[ H \vdash E_1 : \tau_1 \]

\[ H \vdash E_2 : \tau \quad H \triangleright w \leftarrow E_1 \ E_2 \quad \expsem_{HE_1\tau_1} \rho \ t \ 0 = \text{hi} \]

\[ [H \triangleright w : \text{wire } \tau] \rho \ (t + 1) = [H \triangleright E_2 : \text{wire } \tau] \rho \ t \]
Define a substitution mapping $\delta$ as a set of two-tuples:

$$\delta = \{ (w_1, w'_1), \ldots, (w_n, w'_n) \},$$

where $w_i$ and $w'_i$ are wires.

Use $\delta$ to represent a connections to a module’s inputs/outputs.

Define a module instantiation predicate with a new proof rule:

$$\langle \text{module } H (\ldots) \ldots \text{name } H'(\text{domain}(\delta)) \ldots \rangle$$

$$\delta = \{ (w_1, w'_1), \ldots, (w_n, w'_n) \}$$

$$\forall w_i \in \text{domain}(\delta) \quad H \vdash w_i : \tau_i \land H' \vdash w'_i : \tau_i$$

$$H \gg H'(\delta)$$

Read as “$H$ instantiates $H'$ with parameters $\delta$.”

Take $\text{domain}(\delta)$ to be the set \{ $w | \exists w'(w, w' \in \delta)$ \}
Defining Module Instance Semantics

\[(X, X') \in \delta \quad H \gg H'(\delta)\]

\[\forall (w, w') \in \delta [H' \triangleright w' : \text{wire } \tau] \equiv [H \triangleright w : \text{wire } \tau]\]

\[\Rightarrow [H' \triangleright X' : \text{wire } \tau_2] = F\]

\[\therefore [H \triangleright X : \text{wire } \tau_2] = F\]
Decentralized Information Flow Control (DIFC)

- A method for controlling the flow of information in programs
- Developed by Andrew Myers and Barbara Liskov in 2000
- Implemented for JFlow programming languages (Myers)
- Also used in several OS: Asbestos, HiStar, Flume
We define *label* to be a set of tags:
\[ L = \{t_1, t_2, \ldots, t_n\} \]

A *tag* represents some guarantee about the source of information.

Each piece of “information” has two labels associated with it: one for secrecy, and one for integrity.
Secrecy and Integrity

- A secrecy label $L_S$ is a set of tags representing the sources of information that something has possibly seen.
- An integrity label $L_I$ is a set of tags representing the sources of information that endorse the contents of the label's carrier.
- Always safe to add to $L_S$, and remove label from $L_I$.
- Only designated principals (here: modules) may do opposite: declassify data by removing secrecy labels or endorse by adding integrity.
We assign information labels to objects of type bits $n$.

- Can make assertions about all possible bits produced by some wire
  ( = Inference hypotheses)

- Then prove assertions about labels of bits produced by expressions
Inference Rules for Labels

\[ L_S(\llbracket H \triangleright E_1 : \text{wire } n \rrbracket \rho t) = L_{S1} \]
\[ L_S(\llbracket H \triangleright E_2 : \text{wire } n \rrbracket \rho t) = L_{S2} \]
\[ L_I(\llbracket H \triangleright E_1 : \text{wire } n \rrbracket \rho t) = L_{I1} \]
\[ L_I(\llbracket H \triangleright E_2 : \text{wire } n \rrbracket \rho t) = L_{I2} \]
\[ \text{op} \in \{\&, |, \sim\} \quad \quad E = \langle E_1 \text{ op } E_2 \rangle \]

\[ L_S(\llbracket H \triangleright E : \text{wire } n \rrbracket \rho t) = L_{S1} \cup L_{S2} \]
\[ L_I(\llbracket H \triangleright E : \text{wire } n \rrbracket \rho t) = L_{I1} \cap L_{I2} \]
Explained by Myers/Liskov as “leak” through if-stmts
Must handle in Verilog model:
```verilog
if(B) begin ... w <= E ... end
```
Inference rule:
\[
H \triangleright w \leftrightarrow^{E_1} E_2 \quad \quad \quad \quad \quad \quad \quad [H \triangleright E_1 : \text{wire } n_1] = F_1 \\
F_1 \rho t 0 = \text{hi} \quad \quad \quad [H \triangleright E_2 : \text{wire } n_2] = F_2 \\
[H \triangleright w : \text{wire } n] = F \quad \quad \quad L_S(F_1 \rho t) = L_{S1} \\
L_I(F_1 \rho t) = L_{I1} \quad \quad \quad L_S(F_2 \rho t) = L_{S2} \quad \quad \quad L_I(F_2 \rho t) = L_{I2} \\
L_S(F \rho (t + 1)) = L_{S1} \cup L_{S2} \\
L_I(F \rho (t + 1)) = L_{I1} \cap L_{I2}
\]