Lustre → Verilog

Gabriel Desfrene

29 January 2025

The Verilog Language

- Hardware Description Language (HDL) used to model digital circuits.
- Developed in 1984.
- Syntax similar to C.
- Supports multiple paradigms: structural & behavioral.

Example 1

```
module main (
    input wire clock,
    input wire reset,
    input wire x,
    output reg [7:0] y
);
  always @(posedge clock) begin
    if (reset) y <= 8'd0;
    else if (x || y > 0) y <= y + 8'd1;
  end
endmodule
```

Example 2

```
module main (
    input  wire clock,
    input  wire reset,
    input  wire x,
    output reg  y
);
  wire new_y, reset_n, x_or_y;

  not (reset_n, reset);
  or (x_or_y, x, y);
  and (new_y, reset_n, x_or_y);

  always @(posedge clock) begin
    y <= new_y;
  end
endmodule
```

Conclusion

It's verbose and it's ugly.

Objectif

Compiling Lustre into Verilog.

Compiling Lustre into Verilog

- **Why?**
  - Take advantage of Lustre's elegance,
  - Synthesize Lustre models,
  - It looks fun.

Compiling Lustre into Verilog

- **Why?**
  - Take advantage of Lustre's elegance,
  - Synthesize Lustre models,
  - It looks fun.
- **Why Verilog?**
  - Standard language for hardware synthesis,
  - Allows description in terms of logic gates,
  - It's a good opportunity to use it a bit.

Specifications

- Compile a subset of Lustre into *gate-level* Verilog.
- Maintain *retrocompatible* syntax with existing Lustre compilers.
- Add operations to handle buses.

The Lustre Kernel

- Unary operations:
  - `not`: Bool $\rightarrow$ Bool
  - `neg`: Int $\rightarrow$ Int
- Binary operations:
  - `and`, `or`: Bool $\rightarrow$ Bool $\rightarrow$ Bool
  - $+, -$: Int $\rightarrow$ Int $\rightarrow$ Int
  - $=, \neq, <, \leqslant, \geqslant, >$: Int $\rightarrow$ Int $\rightarrow$ Bool
- Branching expressions: Bool $\rightarrow \tau \rightarrow \tau \rightarrow \tau$
- `fby`: $\tau \rightarrow \tau \rightarrow \tau$

Using Bit-Vectors

- Int $:=$ Signed_γ
- Unary operations:
  - `not`: $\mathsf{Bool} \to \mathsf{Bool}$
  - `neg`: $\mathsf{Signed}_\sigma \to \mathsf{Signed}_\sigma$
- Binary operations:
  - `and`, `or`: $\mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool}$
  - $+$: $\tau \to \tau \to \tau$ $\quad\quad \tau \in \{\mathsf{Signed}_\sigma, \mathsf{Unsigned}_\sigma\}$
  - $-$: $\mathsf{Signed}_\sigma \to \mathsf{Signed}_\sigma \to \mathsf{Signed}_\sigma$
  - $=, \neq, <, \leqslant, \geqslant, >$: $\tau \to \tau \to \mathsf{Bool}$ $\quad\quad \tau \in \{\mathsf{Signed}_\sigma, \mathsf{Unsigned}_\sigma\}$
- Branching expressions: $\mathsf{Bool} \to \tau \to \tau \to \tau$
- `fby`: $\tau \to \tau \to \tau$

Bit-Vectors Operation

- `slice` i: $\mathsf{Raw}_\sigma \to \mathsf{Bool}$ $\qquad 0 \leqslant i < \sigma$
- `select` $[i:j]$: $\mathsf{Raw}_\sigma \to \mathsf{Raw}_{j-i}$ $\qquad 0 \leqslant i < j \leqslant \sigma$
- `concat`: $\mathsf{Raw}_{\sigma_1} \mid \mathsf{Bool} \to \mathsf{Raw}_{\sigma_2} \mid \mathsf{Bool} \to \mathsf{Raw}_{\sigma_1+\sigma_2}$
- Conversions: $\mathsf{Raw}_\sigma \to \mathsf{Signed}_\sigma, \mathsf{Signed}_\sigma \to \mathsf{Unsigned}_\sigma, \ldots$
- $=, \neq$: $\mathsf{Raw}_\sigma \to \mathsf{Raw}_\sigma \to \mathsf{Bool}$

Finally

```
node after(x, reset: bool) returns (after: bool);
let
    after = if reset
            then false
            else x or (false fby after);
tel

node main(reset, x: bool) returns (y: u8);
let
    y = 0 fby if after(x, reset) then y + 1 else 0;
tel
```

Finally

```
node after(x, reset: bool) returns (after: bool);
let
    after = if reset
            then false
            else x or (false fby after);
tel

node main(reset, x: bool) returns (y: u8);
let
    y = 0 fby if after(x, reset) then y + 1 else 0;
tel
```

Using Haskell

Rust: Borrow checker required
Haskell: No need, it's immutable!

Rust: Lifetimes are hard
Haskell: Garbage Collector does it for you!

Rust: "Fearless Concurrency"
Haskell: Concurrency is just a Monad!

Rust: Memory safe with effort
Haskell: Memory safe by default!

Using Haskell

Rust: Borrow checker required
Haskell: No need, it's immutable!

Rust: Lifetimes are hard
Haskell: Garbage Collector does it for you!

Rust: "Fearless Concurrency"
Haskell: Concurrency is just a Monad!

Rust: Memory safe with effort
Haskell: Memory safe by default!

Choose Haskell. Be functional. Be safe. Be pure.

Parsing

- We use *Megaparsec*, a *Monadic Parser* to parse our Lustre grammar,
- Monads Used: 2.

Typing

- Constants !!!
- Monads Used: xx.

Normalization

-
- Monads Used: xx.

Verilog Conversion

-
- Monads Used: xx.

Standard Library Design

The Lustre ALU

Examples

Some Stats

Possibles Improvements