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 $\to$ Bool
  - `neg`: Int $\to$ Int
- Binary operations:
  - `and`, `or`: Bool $\to$ Bool $\to$ Bool
  - $+, -$: Int $\to$ Int $\to$ Int
  - $=, \neq, <, \leqslant, \geqslant, >$: Int $\to$ Int $\to$ Bool
- Branching expressions: Bool $\to \tau \to \tau \to \tau$
- `fby`: $\tau \to \tau \to \tau$

Using Bit-Vectors

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

Parsing Tree

```
type Expr = Pos ExprDesc
data ExprDesc
  = ConstantExpr Constant
  | IdentExpr (Pos Ident)
  | UnOpExpr UnOp Expr
  | BinOpExpr BinOp Expr Expr
  | ConvertExpr BitVectorKind Expr
  | ConcatExpr Expr Expr
  | SliceExpr Expr (Int, Int)
  | SelectExpr Expr Int
  | AppExpr (Pos Ident) [Expr]
  | TupleExpr (BiList Expr)
  | IfExpr Expr Expr Expr
  | FbyExpr Expr Expr
  deriving (Show, Eq)

type Pattern = Tree (Pos Ident)
data Equation = Equation Pattern Expr
    deriving (Show, Eq)
```

Typing Tree

```
data TExpr atyp
  = ConstantTExpr Constant atyp
  | VarTExpr VarId atyp
  | UnOpTExpr UnOp (TExpr atyp) atyp
  | BinOpTExpr BinOp (TExpr atyp) (TExpr atyp) atyp
  | IfTExpr VarId (TExpr atyp) (TExpr atyp) atyp
  | ConcatTExpr (TExpr atyp) (TExpr atyp) atyp
  | SliceTExpr (TExpr atyp) (BVSize, BVSize) atyp
  | SelectTExpr (TExpr atyp) BVSize atyp
  | ConvertTExpr (TExpr atyp) atyp
  deriving (Show)

type TArg = Either (TConstant atyp) VarId
data TEquation atyp
  = SimpleTEq VarId (TExpr atyp)
  | FbyTEq VarId (TExpr atyp) (TExpr atyp)
  | CallTEq (NonEmpty VarId) NodeIdent [TArg]
  deriving (Show)
```

Typing

- Constants are typed dynamically:

$$10 : \mathsf{Raw}_{\geqslant 4} \,|\, \mathsf{Unsigned}_{\geqslant 4} \,|\, \mathsf{Signed}_{\geqslant 5}$$

- Equation normalization on the fly.
- Monads Used: 11 (+6 for Typing) (+3 for Causality)

Expression Flattening

```
data CAction
  = SetValCAct CVal
  | UnOpCAct CUnOp CVal
  | BinOpCAct CBinOp CVal CVal
  | IfCAct {ifCond :: VarId, ifTrue :: CVal, ifFalse :: CVal}
  | FbyCAct {initVar :: CVal, nextVar :: CVal}
  | ConcatCAct CVal CVal
  | SliceCAct CVal (BVSize, BVSize)
  | SelectCAct CVal BVSize
  deriving (Show)

data CEquation
  = SimpleCEq CVar CAction
  | CallCEq (NonEmpty CVar) NodeIdent [CVal]
  deriving (Show)
```

Monads Used: 12 (+1).

Verilog Conversion

```
data ModuleInst = ModuleInst
  { name :: ModuleName,
    staticArgs :: [StaticValue],
    controlArgs :: Maybe (ModuleControl Ident),
    inArgs :: [Either Constant Ident],
    outArgs :: NonEmpty Ident
  }
  deriving (Show)

data Expr
  = AssignExpr Ident (Either Constant Ident)
  | InstExpr ModuleInst
  deriving (Show)
```

Monads Used: 12 (+0).

Verilog Output

```
module node_main (
    input wire clock,
    input wire init,
    input wire [7:0] var_x,
    input wire [7:0] var_y,
    output wire [7:0] var_res
);
    lustre_and #(
        .N(8)
    ) call_lustre_and_1 (
        .lhs(var_x),
        .rhs(var_y),
        .res(var_res)
    );
endmodule
```

Monads Used: 13 (+1).

Standard Library Design

```
module lustre_and #(
    parameter N = 1
) (
    input  wire [N-1:0] lhs,
    input  wire [N-1:0] rhs,
    output wire [N-1:0] res
);
  and (res, lhs, rhs);
endmodule
```

```
module lustre_sub #(parameter N = 1)
    ( ... );

  wire [N-1:0] new_rhs;
  not (new_rhs, rhs);

  internal_lustre_adder #(.N(N))
    adder (
      .lhs(lhs), .rhs(new_rhs),
      .carry_in(1'd1), .res(res),
      .flag_Z(), .flag_N(),
      .flag_C(), .flag_V());
endmodule
```

The Lustre ALU

```
module internal_lustre_adder #(parameter N = 1) ( ... );
    wire [N:0] carry;
    assign carry[0] = carry_in;

    genvar i;
    generate
    for (i = 0; i < N; i = i + 1) begin : full_adder_gen
        internal_lustre_bit_adder single_bit_adder (
            .a(lhs[i]), .b(rhs[i]), .carry_in(carry[i]),
            .res(res[i]), .carry_out(carry[i+1]));
    end
    endgenerate

    assign flag_Z = ~(|res);
    assign flag_N = res[N-1];
    assign flag_C = carry[N];
    xor (flag_V, carry[N], carry[N-1]);
endmodule
```

In Short

- $\approx$ 2700 lines of Haskell,
- $\approx$ 300 lines of Verilog,
- $\approx$ 700 lines of C++.

Possibles Improvements

- Enlarge the Lustre core (`pre`, `->`, `merge`, ...),
- Add custom data types and structures,
- Improve Verilog code generation,
- Add buffer registers to reduce the critical path,
- Define a Lustre processor!