Библиотека кода для литературного RTL

Васил Дядов

Среда 25 июля 2018 г.

Содержание

1 Инициализация OCaml сессии

```
#use "topfind";;
#require "hardcaml";;
#require "ppx_deriving_hardcaml";;
#require "ppx_hardcaml";;
```

2 Установка принтера для сигналов

```
let print_signal fmt signal =
  Format.fprintf fmt "%s\n"
    (HardCaml.Signal.Comb.to_string signal);;
#install_printer print_signal;;
```

3 Записыватель сигналов

```
module SignalRecorder (S : HardCaml.Comb.S) =
struct
  type data = (int * S.t) list

  type signal = {name: string; data: data}

  type t =
    { from_time: int
    ; to_time: int
    ; signals: signal list }

  let cut ~from_time ~to_time {name; data} =
    let new_data =
      data
      |> List.filter (fun (t, v) ->
            t > from_time && t <= to_time
         )
    in
    let start_val =
      data
      |> List.find_opt (fun (t, _) ->
            t <= from_time )
    in
    let data =
      match start_val with
      | None -> new_data
      | Some (_, v) ->
          new_data @ [(from_time, v)]
    in
    {name; data}

  module type INTF = sig
    type 'a t

    val t : (string * int) t

    val to_list : 'a t -> 'a list
  end

  module type INTF_IMPL = sig
    module Intf : INTF

    val intf : S.t ref Intf.t
  end

  let of_strings s =
    let signals =
      List.map
```

```
      (fun name -> {name; data= []})
      s
  in
  {from_time= 0; to_time= 0; signals}

let of_interfaces m =
  let signals =
    m
    |> List.map (fun m ->
          let module M = ( val ( m
                                 : (module
                                    INTF)
                               ) ) in
          M.to_list M.t |> List.map fst
        )
    |> List.concat
    |> List.map (fun name ->
          {name; data= []} )
  in
  {from_time= 0; to_time= 0; signals}

let update_signals ~time ~signals ~values =
  signals
  |> List.map
       (fun ({name; data} as signal) ->
         let item =
           List.assoc_opt name values
         in
         match item with
         | None -> signal
         | Some v ->
           match data with
           | (_, old_v) :: _
             when old_v = v ->
               signal
           | _ ->
               { name
               ; data= (time, v) :: data
               } )

module Updater (P : INTF_IMPL) = struct
  let update ?(time_inc= 2) ?time
      {from_time; to_time; signals} =
    let to_time =
      match time with
      | None -> to_time
      | Some t -> t
    in
```

```
    let updates =
      let names =
        P.Intf.to_list P.Intf.t
        |> List.map fst
      and vals =
        P.Intf.to_list P.intf
        |> List.map (fun x -> !x)
      in
      List.map2
        (fun n v -> (n, v))
        names vals
    in
    let signals =
      update_signals to_time signals
        updates
    in
    let to_time = to_time + time_inc in
    {from_time; to_time; signals}

  let update_ref ?time_inc ?time recorder =
    recorder :=
      update ?time_inc ?time !recorder
end

let make_updater i =
  let rec loop i =
    match i with
    | [] -> fun ?time_inc ?time x -> x
    | i :: rest ->
        let module I = ( val ( i
                             : (module
                                INTF_IMPL)
                           ) ) in
        let module U = Updater (I) in
        fun ?time_inc ?time s ->
          s
          |> U.update ?time_inc ?time
          |> loop rest ?time_inc ?time
  in
  let updater = loop i in
  fun ?time_inc ?time ({to_time; _} as s) ->
    let time =
      match time with
      | None -> to_time
      | Some t -> t
    in
    updater ?time_inc ~time s

```

```
  let make_updater_ref i =
    let updater = make_updater i in
    fun ?time_inc ?time recorder ->
      recorder :=
        updater ?time_inc ?time !recorder

  let update ?(time_inc= 2) ?time
      {from_time; to_time; signals} values =
    let to_time =
      match time with
      | None -> to_time
      | Some t -> t
    in
    let signals =
      update_signals ~time:to_time ~signals
        ~values
    in
    let to_time = to_time + time_inc in
    {from_time; to_time; signals}

  let update_ref ?time_inc ?time recorder
      values =
    recorder :=
      update ?time_inc ?time !recorder
        values

  let get_view ~from_time ~to_time
      {signals; _} =
    { from_time
    ; to_time
    ; signals=
        signals
        |> List.map
             (cut ~from_time ~to_time) }
end
```

4 Генератор временных диаграмм

```
module TimingDiagram (S : HardCaml.Comb.S) =
struct
  let header =
    <<timing_diagram_header>>

  let footer ~periods =
    <<timing_diagram_footer>>
```

```
8
9   module Recorder = SignalRecorder (S)
10
11  type format = Bin | Dec | Hex
12
13  type spec = {fmt: format}
14
15  type specs = (string * spec) list
16
17  let default_spec = {fmt= Hex}
18
19  let draw_signal ~spec:{fmt; _} ~from_time
20      ~to_time ~signal:{Recorder.name; data} =
21    match data with
22    | [] -> "U"
23    | _ ->
24        let width =
25          S.width (snd (List.hd data))
26        in
27        let string_of_value v =
28          match width with
29          | 1 ->
30              if v = S.vdd then "h" else "l"
31          | _ ->
32              let open Printf in
33              let v = S.to_int v in
34              ( match fmt with
35              | Bin -> sprintf "d{%d'd%d}"
36              | Dec -> sprintf "d{%d'd%d}"
37              | Hex -> sprintf "d{%d'x%X}"
38              )
39                width v
40        in
41        let times = List.map fst data in
42        let timediffs =
43          let open List in
44          map2 ( - ) (to_time :: times)
45            (rev (from_time :: rev times))
46          |> rev
47        in
48        let first_timediff, timediffs =
49          List.(hd timediffs, tl timediffs)
50        in
51        let values =
52          let open List in
53          data |> map snd
54          |> map string_of_value
55          |> rev
56        in
```

```
      let timediffs, values =
        if first_timediff = 0 then
          (timediffs, values)
        else
          ( first_timediff :: timediffs
          , "u" :: values )
      in
      let waveform =
        List.(
          map2 ( ^ )
            (map string_of_int timediffs)
            values)
        |> String.concat " "
      in
      name ^ "&" ^ waveform ^ "\\\\\n"

let out_signals ~clock_name ~default_spec
    ~specs
    ~recorder:{ Recorder.from_time
              ; to_time
              ; signals } =
  let periods = to_time - from_time in
  clock_name ^ "& "
  ^ string_of_int periods
  ^ "{c}\\\\\n"
  ^ ( signals
    |> List.map
         (fun ( {Recorder.name; data} as
              signal )
         ->
           let spec =
             List.assq_opt name specs
           in
           let spec =
             match spec with
             | None -> default_spec
             | Some spec -> spec
           in
           draw_signal ~spec ~from_time
             ~to_time ~signal )
    |> String.concat "" )
  ^ footer ~periods:(periods / 2)

let gen_latex ?(clock_name= "CLK")
    ?(default_spec= default_spec)
    ?(specs= []) recorder =
  header
  ^ out_signals ~clock_name ~default_spec
```

```
        ~specs ~recorder
end
```

4.1 Тест генератора временных диаграмм

```
let _ =
  let module B = HardCaml.Bits.Comb.
                 IntbitsList in
  let module TD = TimingDiagram (B) in
  let module R = TD.Recorder in
  let recorder =
    R.of_strings ["clear"; "data"]
  in
  List.fold_left R.update recorder
    [ [ ("clear", B.constb "1")
      ; ("data", B.constb "0010") ]
    ; [ ("clear", B.constb "1")
      ; ("data", B.constb "0001") ]
    ; [ ("clear", B.constb "0")
      ; ("data", B.constb "1000") ]
    ; [ ("clear", B.constb "0")
      ; ("data", B.constb "0101") ] ]
  |> TD.gen_latex
  |> Printf.printf "%s\n%!"
```