Automatically Generation of N-bit Logic Circuit Components in Extended LOTOS from High-Level Functional Programming Language: HDCaml

Graduate School of Science and Technology,
Shinsu University

09TA684A

Pratima K. Shah

Outline

- Background and Purpose
- Workflow Diagram from HDCaml to LOTOS
- DILL - Digital Logic Circuit Library
- Formal Description Language and High-level Language
- Example of HDCaml code and Generated LOTOS code
- Case Study
- Conclusion and Future Task

Background and Purpose

Background

- HDL (Hardware Description Language) and Formal Verification
- Formal Description Technique
  - → LOTOS
  - → E-LOTOS
  - → DILL (Digital Logic in LOTOS)
- High-Level Language
  - → HDCaml
  - → Ocaml (Objective Caml)
- Code Generation
  - → Technique to generate target code from source code for hardware implementation and verification

Purpose of the study

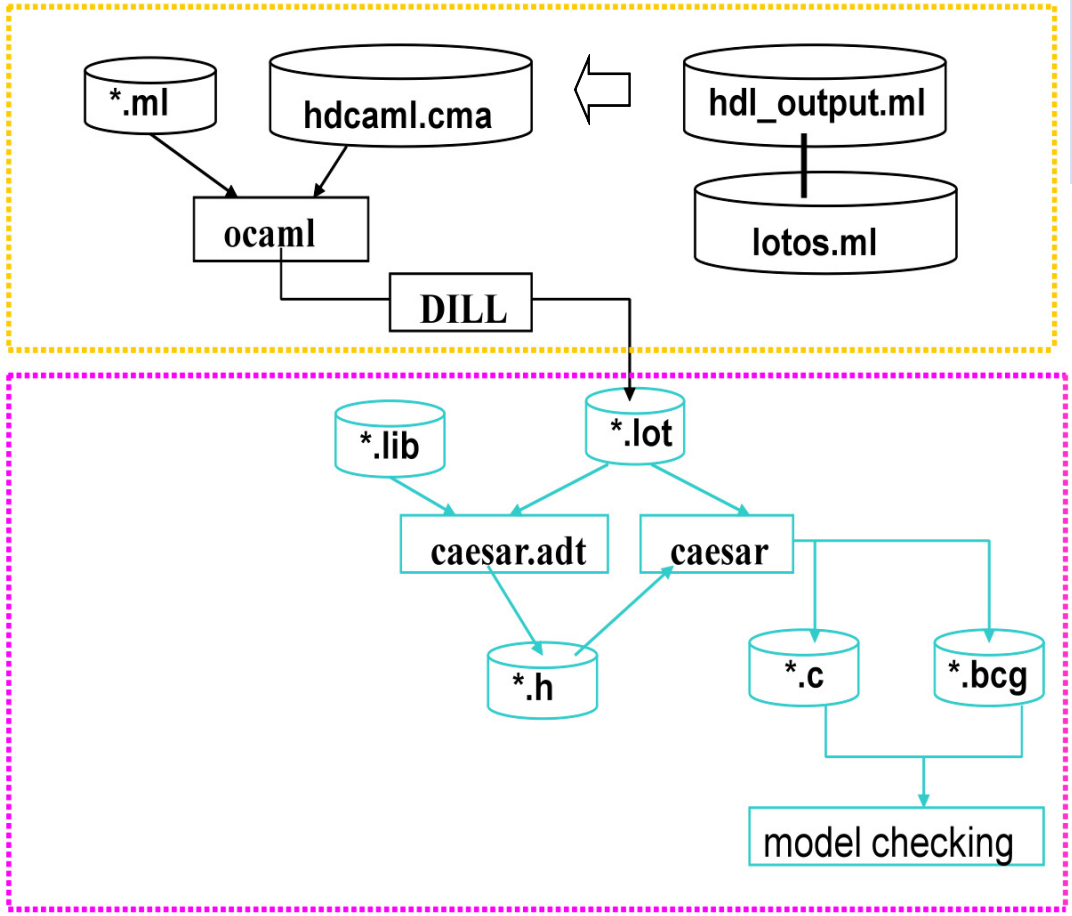
1. To extend the HDL code generator module to map DILL library component in order to generate the Formal Specification Code for Hardware implementation and Formal verification.

2. Code generation reduces the coding cost and time.

3. To apply DILL Data types for the purpose to split and concat the BitArray (Bus) into an individual Bit (Wire) and vice versa in the generated code during model checking.

work-flow diagram From HDCaml to LOTOS

Work flow diagram from HDCaml to LOTOS code



- The HDCaml library archive and the circuit description code are compiled and the LOTOS code is generated.
- The generated LOTOS code is used for the verification of the system by CADP Toolbox.
- CADP is a popular Toolbox developed by the VASY team at INRIA .

DILL (Digital Logic in LOTOS)

- Library of Specification Description Language (J. K Turner/ 1999)
- Here circuit is described by m4 macro language
- Basic elementary elements are AND, OR and Not gate.
- Features of Black box and White box.

Component	White box	Black box
encoder/decoder	○	○
comparator	○	○
parity checker	○	○
Mux/ demux	○	○
latch	○	○
flipflop	○	○
counter	○	×
register	○	×
memory	○	×

Macros for describing circuits in DILL

MWire and MComp

MComp - reproduction of circuit component

MWire - reproduction of connected wire.

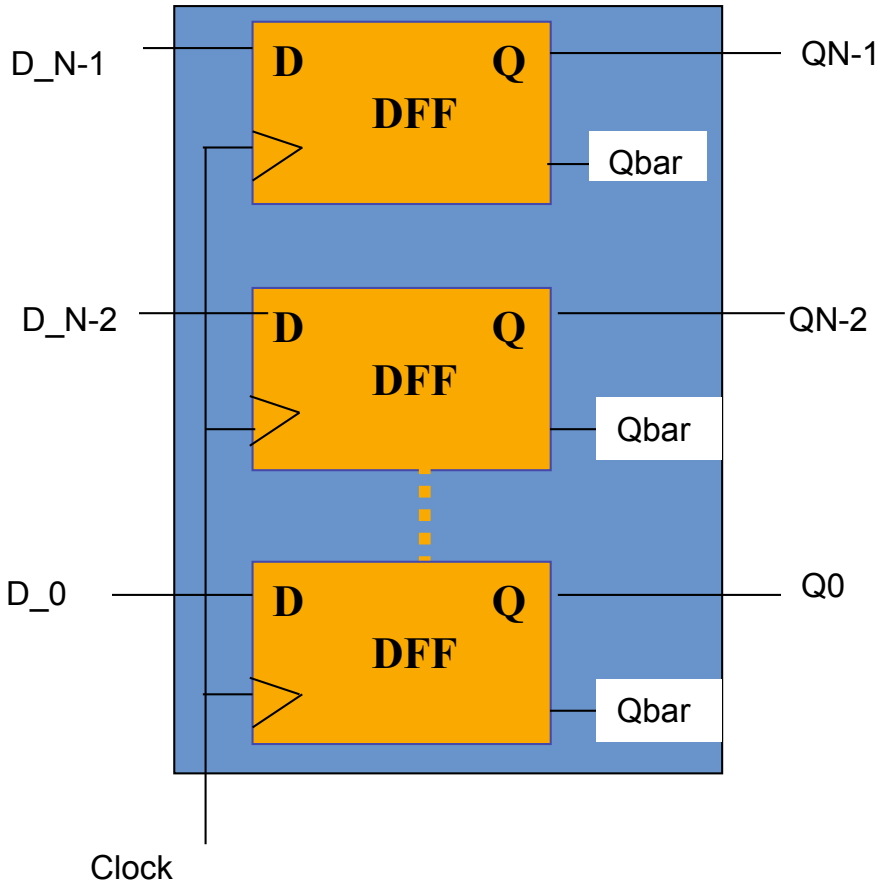
example:

“Register_N_Decl” defines N-bit register in DILL Library :

```
MWire(BIT_W,`D')

MWire(BIT_W, `Q, Qbar')

MComp(BIT_W ,Clk=, `DFlipFlop
[D, Clk=, Q, Qbar]')
```



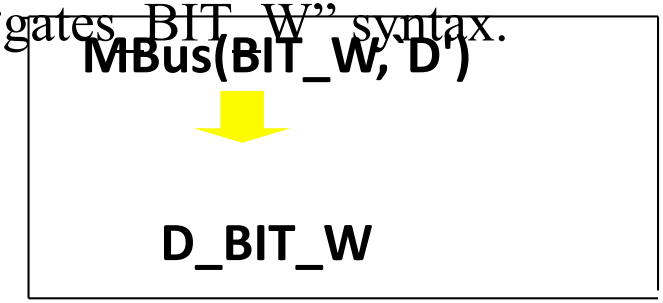
MBus (Bus containing Group of Wires)

- But, in the digital circuits it is quite usual that some related signals are regarded as a whole.

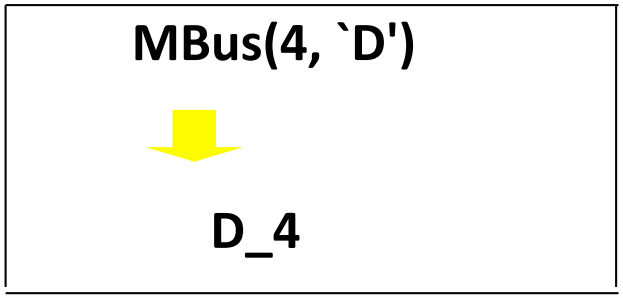
- For example, in a computer system a group of 16-bit data signals could be seen as one signal carried by a data bus.

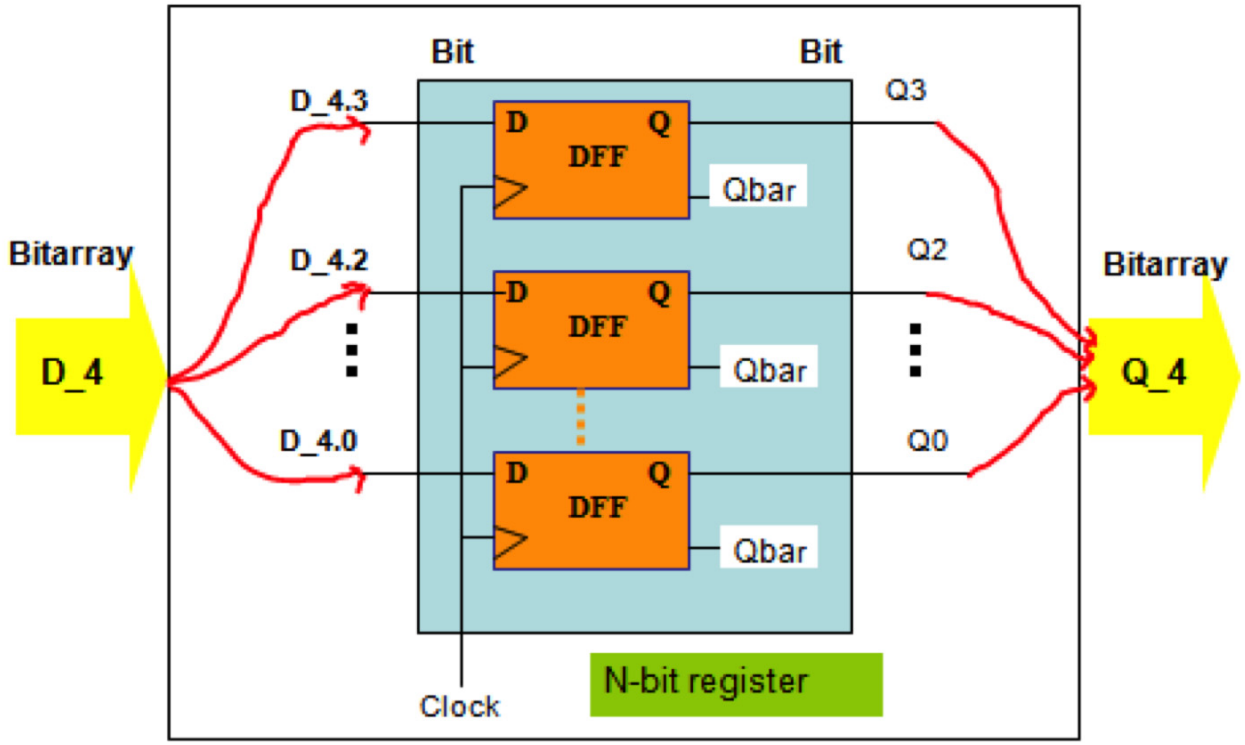
Using the term from computer hardware, we refer to a wire carrying the multi-bit signal as a *bus*.

For this, we wrote another macro definition named MBus which is supported by the m4 macro library, being ultimately translated into a LOTOS specification as "gates_BIT_W" syntax.



For example





Note: MBus depends on bitarray data type

```
define((`Register_N_Decl', `define ((`BIT_W',  $1)‘
   `declare(`$0$1',`DFlipFlop_Decl
  process   `Register_' BIT_W [MBus(BIT_W,`D'),  Clk,
MBus(BIT_W, `Q, Qbar')] :noexit:=
        hide MWire(BIT_W,Qbar) in
       MComp(BIT_W ,Clk=, `DFlipFlop [D, Clk=, Q, Qbar]')
  endproc (* `Register_' BIT_W *)
  ')')
```

DILL Data Type Bitarray operator

Bitarray operator

concates a bit array and bit, or a bit and a bit array to form new bit array

concates two bitarrays to form a new bitarray

. Gets the value of a particular bit from a bitarray.

```
……..   process Register_4Aux[D_4, Clk, Q_4] (dtD_4 : BitArray, dtClk : Bit, dtQ_4 : BitArray) noexit :=
     (D_4 ? dtD_4 : BitArray;
      Register_4Aux[D_4, Clk, Q_4] (dtD_4, dtClk, dtQ_4)
 [ ]   Clk ? newdtClk : Bit;
        ( [(dtClk ne 0 of Bit) or (newdtClk ne 1 of Bit)] ->
         Register_4Aux[D_4, Clk, Q_4] (dtD_4, dtClk, dtQ_4)
 [ ]
          [(dtClk eq 0 of Bit) or (newdtClk eq 1 of Bit)] ->
        (Let dtD_4 : BitArray = Bit (dtD_4.3)#(dtD_4.2)#(dtD_4.1)#(dtD_4.0)   in
         Register_4Aux[D_4, Clk, Q_4] (dtD_4, dtClk, dtQ_4) ) )
 [ ]
      Q_4 ! dtQ_4;
       ( Let dtQ_4 : BitArray=(((Bit (1)#1)#0)#0) in
         Register_4Aux[D_4, Clk, Q_4] (dtD_4, dtClk, dtQ_4))))……..
```

High-level Language: HDCaml

- High-level language, **HDCaml** is a functional language for generic hardware description at RTL (Register Transfer Level). This HDL is embedded in the functional Objective Caml (OCaml) language that contains a code generator for existing HDLs
- HDCaml is an open source.
- Output : verilog, VHDL netlist, System C, etc.
- Circuit Types

→ type signal = Circuit.signal Signals represent bit vectors.

→ type circuit = Circuit.circuit A circuit is a completed circuit design.

Example of HDCaml is shown in slide no. 14

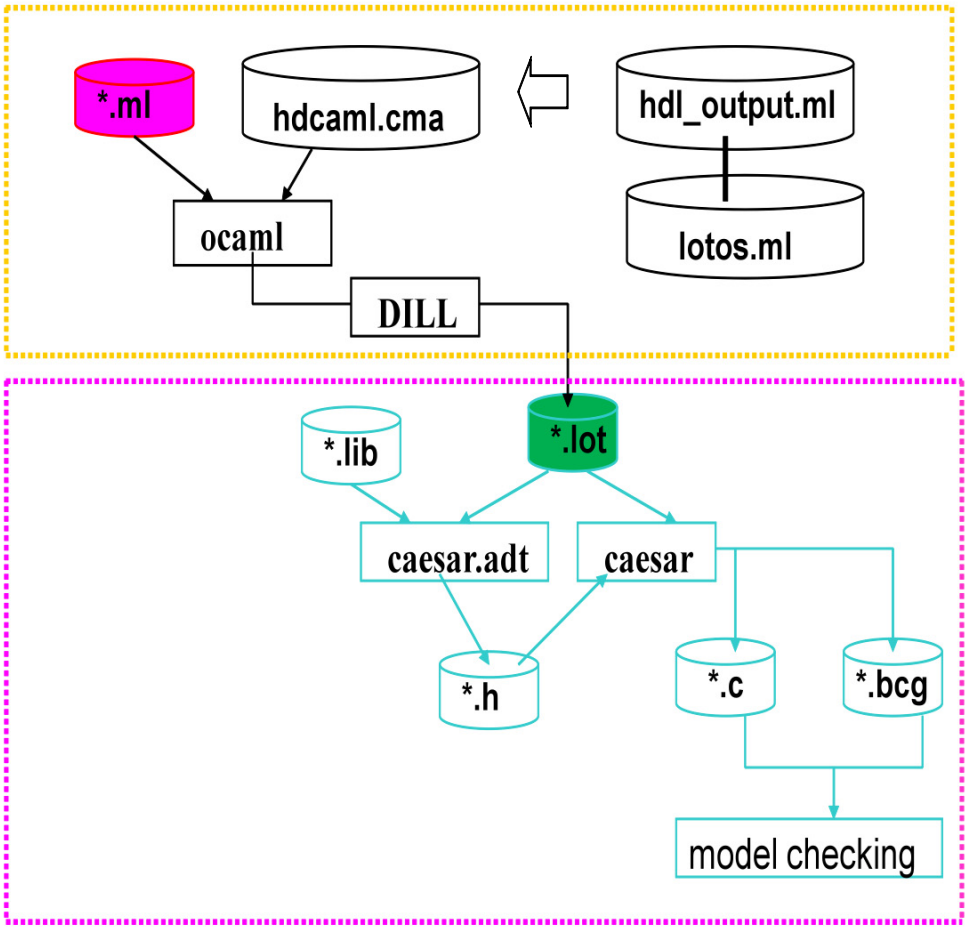
Internal Circuit Representation

In HDCaml, to interpret the described logic circuit composition, and to use it for the output generator function to generate output code , the data type ie. Circuit data type is defined.

```
type circuit = Circuit of id * string * circuit list * signal list * sink list
…
and signal
= Signal_input of id * string * width
| Signal_signal of  id * string * width * signal ref
| Signal_const of  id * string
| Signal_empty of id
| Signal_select of  id * int * int * signal
| Signal_concat of id * signal * signal
| Signal_not                                  of id * signal
| Signal_and                                  of id * signal * signal
| Signal_xor                                  of id * signal * signal
| Signal_or                                   of id * signal * signal
| Signal_eq                                   of id * signal * signal
| Signal_lt                                   of  id * signal * signal
| Signal_add                                  of id * signal * signal
| Signal_sub                                  of id * signal * signal
| Signal_mul_u of id * signal * signal
| Signal_mul_s of id * signal * signal
| Signal_mux                                  of id * signal * signal * signal
| Signal_reg                                  of id * signal * signal
…
;;
```

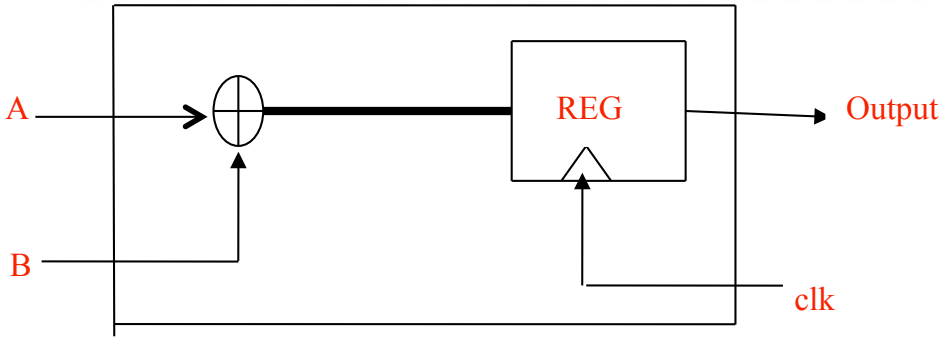
Simple Example of Generated LOTOS Code from HDCaml

Work flow diagram from HDCaml to LOTOS code



- Pink one is the HDCaml code
- Green one is the generated LOTOS code

Example of HDCaml and generated LOTOS code



#1.

#1. Circuit connection of addition of 2 nput
#2. HDCaml code of the circuit connection
#3. Generated LOTOS code from HDCaml

```
open Hdcaml;;
 open design;;
let reg_module enable input1 input2 =
     let output = reg enable (input1 +: input2) in
      output
let reg_module_design input_width =
  start_circuit “reg_module”;
  let enable = input “enable” 1 in
  let input1 = input “A” input_width in
  let input2 = input “B” input_width in
  output “output” (reg_module enable input1 input2);
let circuit = get_circuit () in
Lotos.output_netlist circuit;;;
reg_module_design 4;;
```

#2.

```
specification reg_module [reset, clock, A, B,
enable, output]:  noexit
library
       OpenDill
  endlib
behaviour
   reg_module [reset, clock, A, B, enable, output]
Where
   process reg_module [A, B, enable, output] :
noexit :=
    hide output,  n_13 in
      (
       Register_4  [enable, n_13, output]
       |[n_13]|
       FullAdder_4 [A, B, n_13]
       )
   endproc
 endspec
```

#3

Peculiar problem to LOTOS code generation

1. Definition of internal connected line

If the circuit is connected to the multiple devices (sub circuit) within it, it is necessary to describe the internal signal/wire connecting the sub circuit.

2. Correspondence to RTL description

In HDCaml, the number of bits (BitWidth/BitArray) can be freely defined, because of rich in different operators. However, it is difficult to directly generate code of N-bit in LOTOS.

3. Process synchronization

In HDCaml, there is no clear process synchronization, as circuit described here is at RTL level, where as in the process definition by LOTOS, each operation process should describe the synchronizing gate specifying it.

For example, … output = ((A &: B) |: C) ….. hide signal

And2[A, B, signal] |[signal]|

Or2 [C, signal, output]…….

By extending the HDL code generator module and solving all these three problems step by step, it was possible to generate the N-bit components in E-LOTOS code from high-level HDCaml.

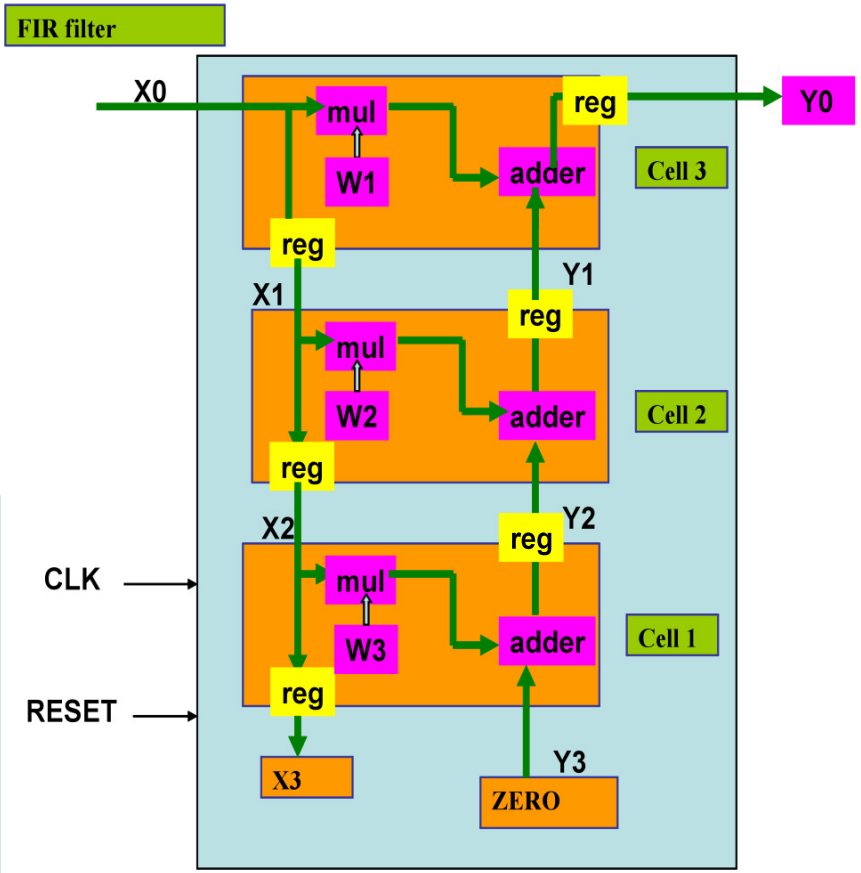
Case Study 1

HDCaml Code of cell_module

```
……..let cell_module enable yin xin coeff ( *+ ) =
let cell_output = reg enable (yin +: ( xin *: coeff) ) in
cell_output    ;;
let cell_module_design xin_width coeff_width yin_width =
      start_circuit “cell_module”;
      let enable = input “enable” 1 in
      let x0 = input “x0” xin_width in
      let x1 = signal “x1” xin_width in
      x1 <== “x1_reg” – enable x0;
      let x2 = signal “x2” xin_width in
      x2 <== “x2_reg” – reg enable x1;
      let x3 = signal “x3” xin_width in
      x3 <== “x3_reg” – reg enable x2;
     let y3 = input “zero” yin_width in
      let coeff3 = input “w3” coeff_width in
      let y2 = signal “y2” yin_width in
      y2 <== (cell_module enable y3 x2 coeff3(*+));
      let coeff2 = input “w2” coeff_width in
      let y1 = signal “y1” yin_width in
      y1 <== (cell_module enable y2 x1 coeff( *+ ));
      let coeff1 = input “w3” coeff_width in
      output “y0” (cell_module enable y1 x0 coeff ( *+ ));
       let circuit = get_circuit () in
       Lotos.output_netlist circuit;    ;;
      cell_module_design 2 2 4;;
```

Each Internal composition is described as function

The bitwidth is specified at call

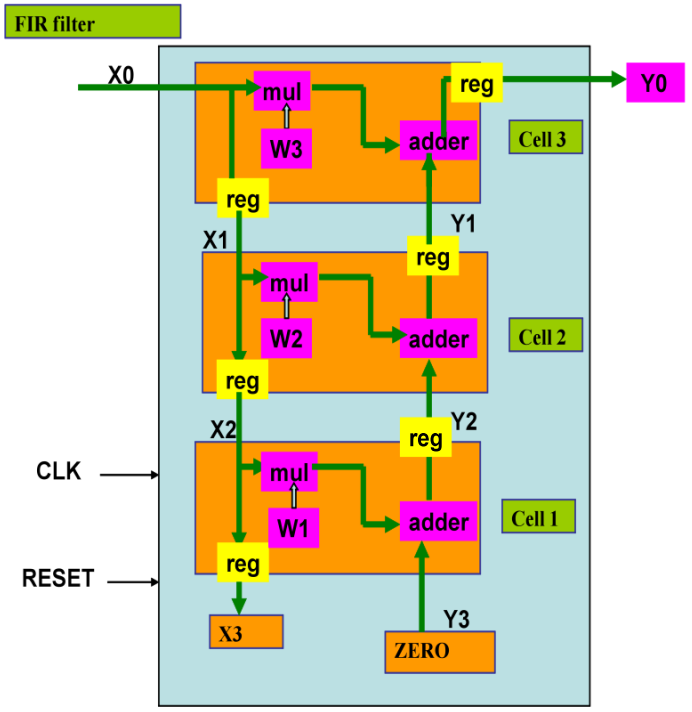


Circuit connection of Cell module (fixed for 3-cells).

Generation of Internal connected line in LOTOS

```
let x1 = signal "x1" xin_width in
 x1 <== "x1_reg" – enable x0;
 let x2 = signal "x2" xin_width in
 x2 <== "x2_reg" – reg enable x1;
 let x3 = signal "x3" xin_width in
 x3 <== "x3_reg" – reg enable x2;
.....
 let y2 = signal "y2" yin_width in
y2 <== (cell_module enable y3 x2 coeff3(*+));
.....
 let y1 = signal "y1" yin_width in
y1 <== (cell_module enable y2 x1 coeff( *+ ));
```

- Definition of internal connected line in a circuit is generated in between hide ~ in



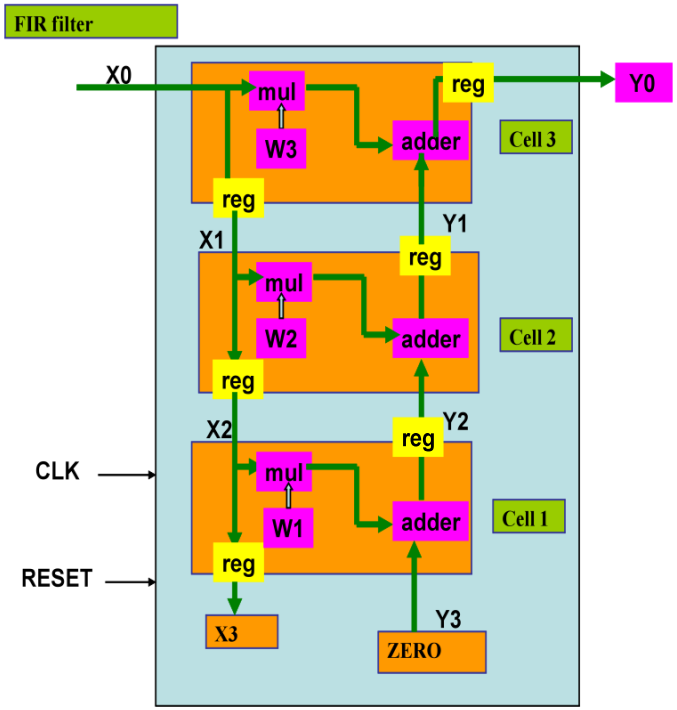

```
.....
hide y1, n_34, n_33, y1_reg, y2, n_30, n_29, y2_reg,
n_25, n_24, x2, x3_reg, x1, x2_reg, x1_reg in
 ..........
```

Each wires behaves as a BUS

Generation of Bit Operation in LOTOS Code

- Correspondence to RTL circuit description , LOTOS is described by Bit operation
- Bit Operation module use definition of DILL

```
……..
  hide (*internal bus*) in
  ((Register_4 [enable, n_34, y0]) |[enable, n_34]|
    ((FullAdder_4 [y1, n_33, n_34]) |[n_33]|
      ((Multiplier_2_4 [x0, w3, n_33]) |[x0]|
        ((Register_4 [enable, n_30, y1_reg]) |[enable, n_30]|
          ((FullAdder_4 [y2, n_29, n_30]) |[n_29]|
            ((Multiplier_2_4 [x1, w2, n_29]) |[x1]|
              ((Register_4 [enable, n_25, y2_reg]) |[enable, n_25]|
                ((FullAdder_4 [zero, n_24, n_25]) |[n_24]|
                  ((Multiplier_2_4[x2, w1, n_24]) |[x2]|
                    ((Register_2 [enable, x2, x3_reg]) |[enable]|
                      ((Register_2 [enable, x1, x2_reg]) |[enable]|
                        (Register_2 [enable, x0, x1_reg])
                         ))))))))))
……..
```



Generation of Process synchronization in LOTOS

```
……..
  hide (*internal bus*) in
  ((Register_4 [enable, n_34, y0]) |[enable, n_34]|
   ((FullAdder_4 [y1, n_33, n_34]) |[n_33]|
     ((Multiplier_2_4 [x0, w3, n_33]) |[x0]|
      ((Register_4 [enable, n_30, y1_reg]) |[enable, n_30]|
       ((FullAdder_4 [y2, n_29, n_30]) |[n_29]|
        ((Multiplier_2_4 [x1, w2, n_29]) |[x1]|
        ((Register_4 [enable, n_25, y2_reg]) |[enable, n_25]|
         ((FullAdder_4 [zero, n_24, n_25]) |[n_24]|
           ((Multiplier_2_4[x2, w1, n_24]) |[x2]|
            ((Register_2 [enable, x2, x3_reg]) |[enable]|
              ((Register_2 [enable, x1, x2_reg]) |[enable]|
               (Register_2 [enable, x0, x1_reg])
                  ))))))))))
……..
```

- There must be synch-ronization wire in betw-een two processes in case of LOTOS.
- The synchronization line does not appear clearly in case of HDCaml.
- Here each wires behaves as a Bus instead of behaving as like a single wire.

Generated LOTOS code output showing Internal Composition, RTL description, and Process synchronization

```
……..
 process cell_module [w3, w2, w1, zero, x0, enable, y0] : noexit :=
     hide y1, n_34, n_33, y1_reg, y2, n_30, n_29, y2_reg, n_25, n_24, x2, x3_reg, x1, x2_reg,
x1_reg in
       ((Register_4 [enable, n_34, y0]) |[enable, n_34]|
         ((FullAdder_4 [y1, n_33, n_34]) |[n_33]|
          ((Multiplier_2_4 [x0, w3, n_33]) |[x0]|
            ((Register_4 [enable, n_30, y1_reg]) |[enable, n_30]|
              ((FullAdder_4 [y2, n_29, n_30]) |[n_29]|
                ((Multiplier_2_4 [x1, w2, n_29]) |[x1]|
                  ((Register_4 [enable, n_25, y2_reg]) |[enable, n_25]|
                    ((FullAdder_4 [zero, n_24, n_25]) |[n_24]|
                      ((Multiplier_2_4[x2, w1, n_24]) |[x2]|
                        ((Register_2 [enable, x2, x3_reg]) |[enable]|
                          ((Register_2 [enable, x1, x2_reg]) |[enable]|
                            (Register_2 [enable, x0, x1_reg])
                           ))))))))))
   endproc
……..
```

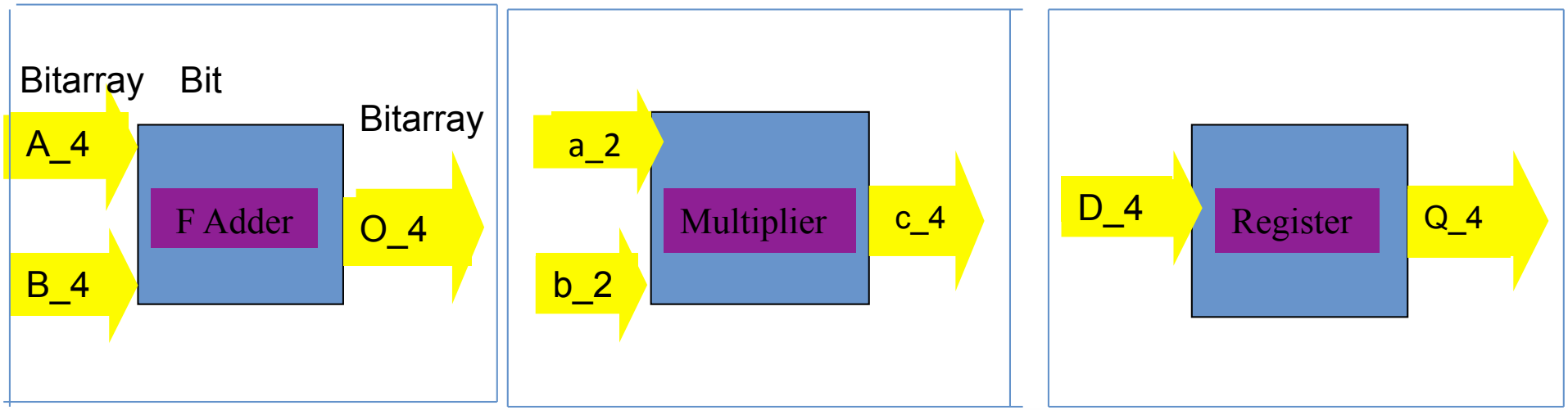
1. Green color shows Global input.

2. Blue color shows internal connected wire.

3. Red color shows Global output.

Structural and Behavioral Expression of circuit

- In this study, the structural specification of the circuit design has generated from high-level hardware description language HDCaml
- Every model has structural (Connection between component and wire) and behavioral specification (what happens between the inputs and outputs).
- The BB behavioral specification of the generated components (Register, Multiplier, and Adders) is coded by hand into the generated LOTOS file to test whether the automatically generated code works correctly within the specification.



BB Behavioral Expression Register component

```
……..
process Register_4[D_4, Clk, Q_4] : noexit :=
        Register_4Aux[D_4, Clk, Q_4]
        ( Bit (X)#X#X#X,              (* dtD_4.(0..3)    : 4Bit*)
          X of Bit,                   (*dtClk          : 1Bit*)
          Bit (X)#X#X#X               (*dtQ_4.(0..3)    :4Bit *)
where
   process Register_4Aux[D_4, Clk, Q_4] (dtD_4 : BitArray, dtClk : Bit, dtQ_4 : BitArray) noexit :=
      (D_4 ? dtD_4 : BitArray;
       Register_4Aux[D_4, Clk, Q_4] (dtD_4, dtClk, dtQ_4)
  [ ]   Clk ? newdtClk : Bit;
         ( [(dtClk ne 0 of Bit) or (newdtClk ne 1 of Bit)] ->
          Register_4Aux[D_4, Clk, Q_4] (dtD_4, dtClk, dtQ_4)
  [ ]
         [(dtClk eq 0 of Bit) or (newdtClk eq 1 of Bit)] ->
          (Let dtD_4 : BitArray = Bit (dtD_4.3)#(dtD_4.2)#(dtD_4.1)#(dtD_4.0)   in
           Register_4Aux[D_4, Clk, Q_4] (dtD_4, dtClk, dtQ_4) ) )
  [ ]
       Q_4 ! dtQ_4;
         ( Let dtQ_4 : BitArray=(((Bit (1)#1)#0)#0) in
           Register_4Aux[D_4, Clk, Q_4] (dtD_4, dtClk, dtQ_4))))
      ……..
```

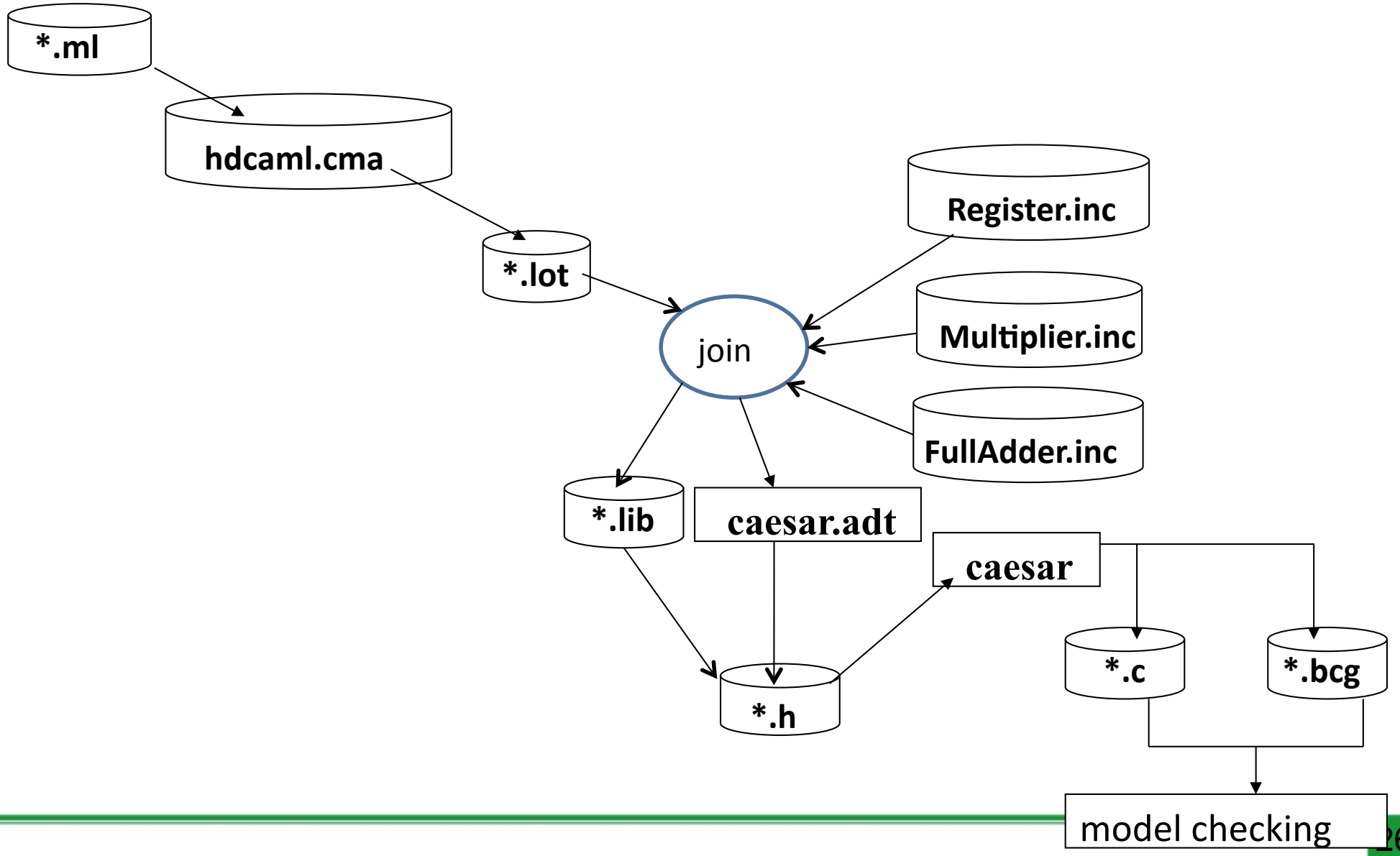
BB Behavioral Expression of Multiplier component

```
……
  process Multiplier_2_4[a_2, b_2, c_4] : noexit :=
        Multiplier_2_4Aux[a_2, b_2, c_4]
        ( Bit (X)#X,              (* dta_2.(0..1)   : 2Bit*)
         Bit (X)#X,               (* dtb_2.(0..1)   : 2Bit*)
         Bit (X)#X#X#X            (* dtc_4.(0..3)   :4Bit *)
  where
  process Multiplier_2_4Aux[a_2, b_2, c_4] (dta_2 : BitArray, dtb_2 : BitArray, dtc_4 : BitArray) : noexit :=
    (a_2 ? dta_2 : BitArray;
     Multiplier_2_4Aux[a_2, b_2, c_4] (dta_2, dtb_2, dtc_4)
   [ ]
     b_2 ? dtb_2 : BitArray;
     Multiplier_2_4Aux[a_2, b_2, c_4] (dta_2, dtb_2, dtc_4))
  [ ]
     (c_4 ! dtc_4;
       (Let dtc_4 : BitArray = (((Bit (1)#1)#0#0) mul (((Bit (1)#1)#1)#1) in
        Multiplier_2_4Aux[a_2, b_2, c_4] (dta_2, dtb_2, dtc_4)))
    ……..
```

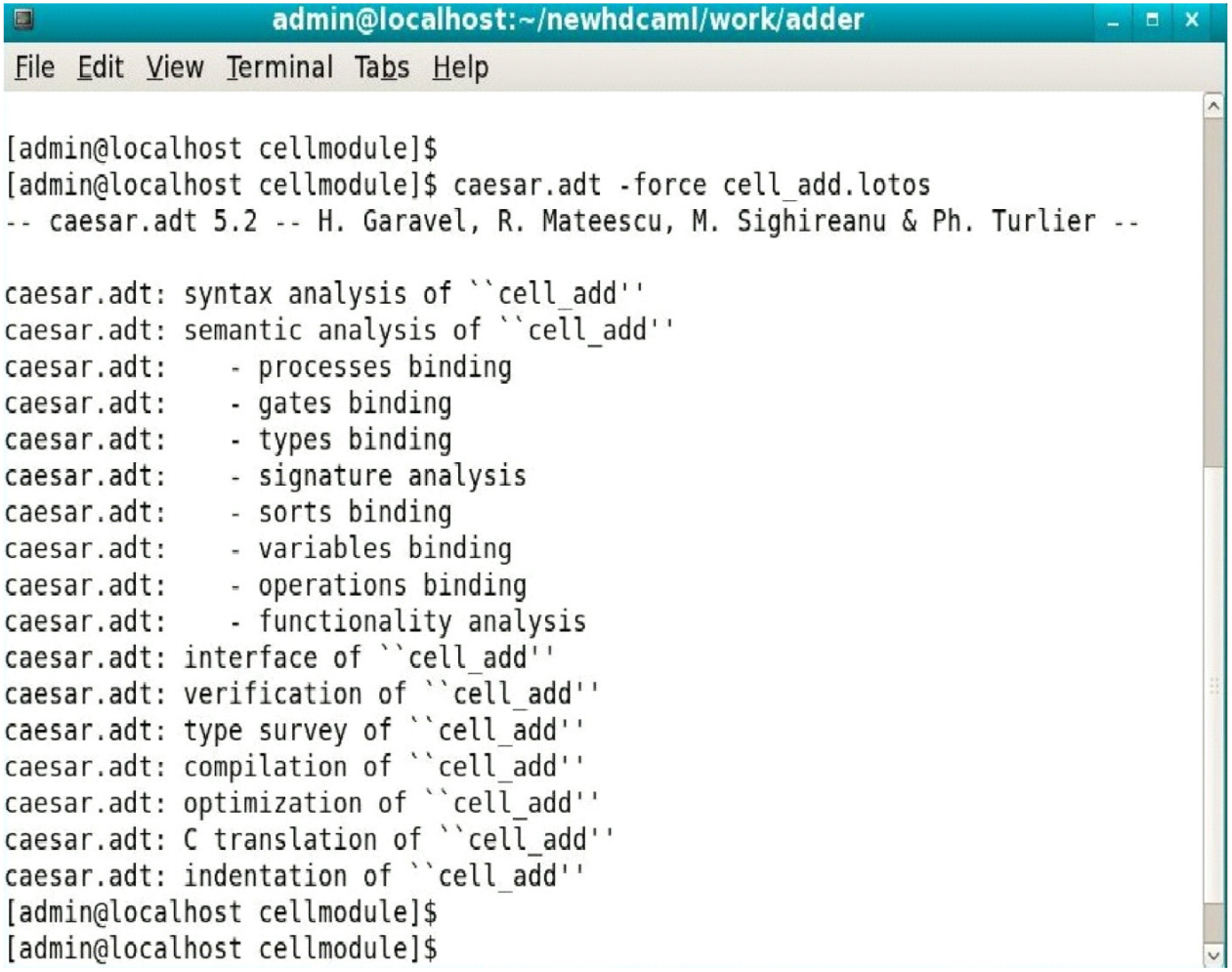
BB Behavioral Expression of Full Adder component

```
……..
process FullAdder_4[A_4, b_4, O_4] : noexit :=
        FullAdder_4Aux[A_4, B_4, O_4]
        ( Bit (X)#X#X#X,            (* dtA_4.(0..3)   : 4Bit*)
         Bit (X)#X#X#X,             (* dtB_4.(0..3)   : 4Bit*)
         Bit (X)#X#X#X              (* dtO_4.(0..3)   :4Bit *)
  where
  process FullAdder_4Aux[A_4, B_4, O_4] (dtA_4 : BitArray, dtB_4 : BitArray, dtO_4 :
BitArray) : noexit :=
     (A_4 ? dtA_4 : BitArray;
      FullAdder_4Aux[A_4, B_4, O_4] (dtA_2, dtB_4, dtO_4)
    [ ]
      B_4 ? dtB_4 : BitArray;
      FullAdder_4Aux[A_4, B_4, O_4] (dtA_4, dtB_4, dtO_4))
   [ ]
      ( O_4 ! dtO_4;
        (let dtO_4 : BitArray = (((Bit (1)#1)#0#0) Add (((Bit (1)#1)#1)#1) in
         FullAdder_4Aux[A_4, B_4, O_4] (dtA_4, dtB_4, dtO_4)))
……..
```

Generated Structural LOTOS Code + Added Behavioral LOTOS code



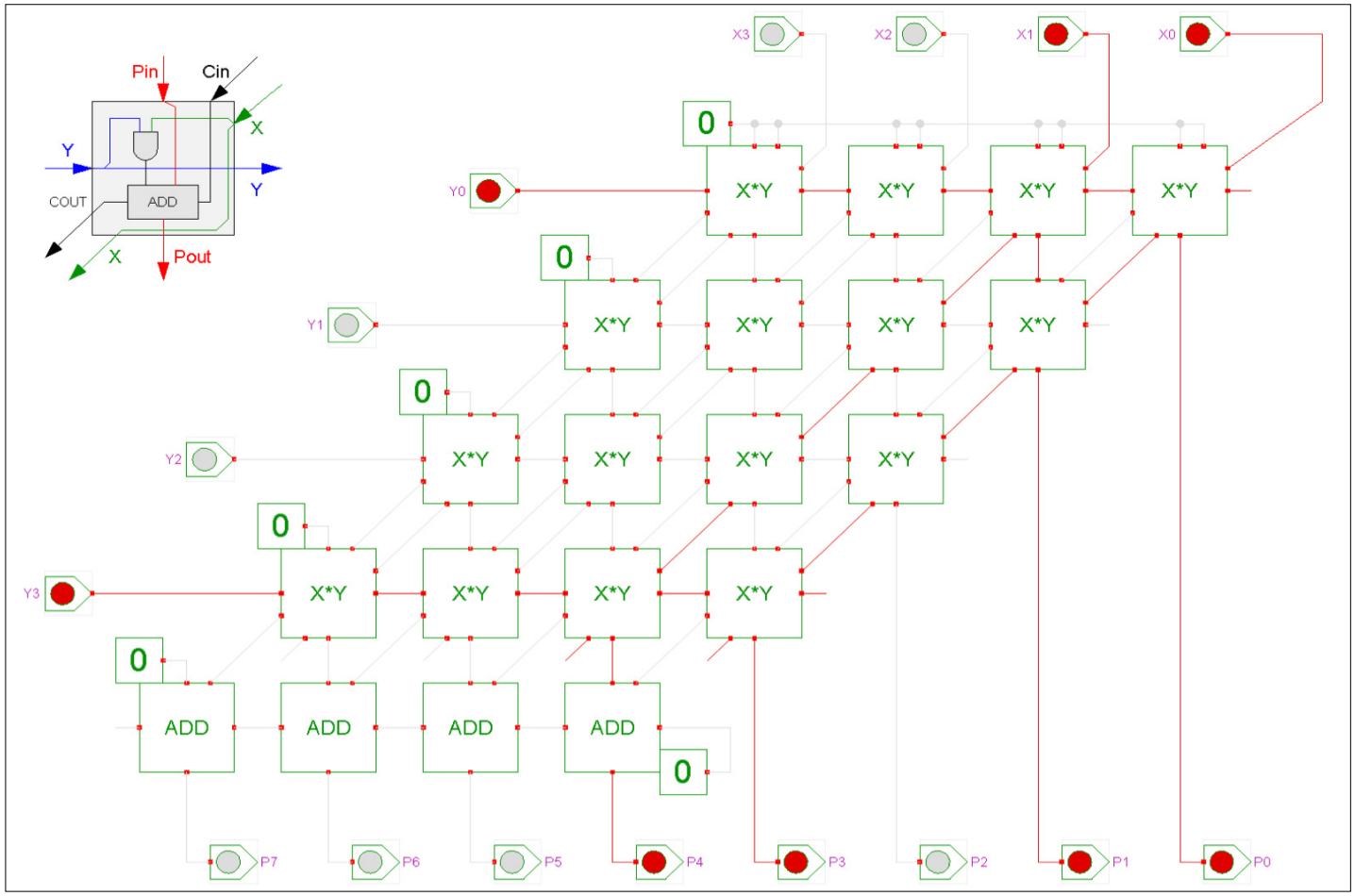
Verification of Generated LOTOS Code



- The N-Bit Generated LOTOS code from HDCaml and added behavioral code was verified.

Case Study 2

4*4 Multiplier Circuit



N*N Circuit connection diagram of parallel multiplier

HDCaml and Generated LOTOS code of Multiplier circuit

```
let multiplier x_l y_l=
…
let rec x_loop x_l y p_l c_l=
match x_l with
| [] -> [zero 1],[]
| x::x_r-> let bit_mul= x &: y in
      let p_out,c_out= full_adder bit_mul
                        (List.hdp_l) (List.hdc_l) in
     let p_r,c_r = x_loopx_ry (List.tlp_l) (List.tlc_l) in
                        p_out::p_r,c_out::c_r
     in
let rec y_loop x_l y_l p_l c_l =
     match y_l with
       | [] -> ripple_adderp_lc_l(zero 1)
       | y::y_r->
           let p_new, c_new= x_loop x_l y p_lc_l in
           (List.hdp_new)
          ::(y_loopx_ly_r(List.tlp_new) c_new)
 in
y_loopx_ly_l(ini_listx_len) (ini_listx_len)
;;
```

```
... …(Xor2 [n_39, n_45, output_1] |[n_39, n_45]|
       (Xor2 [n_37, n_27, n_45] |[n_37, n_27]|
       (Or2 [n_43, n_40, n_44] |[n_43, n_40]|
       (Or2 [n_42, n_41, n_43] |[n_42, n_41]|
      (And2 [n_39, n_37, n_42] |[n_39, n_37]|
      (And2 [n_37, n_27, n_41] |[n_37, n_27]|
      (And2 [n_27, n_39, n_40] |[n_27, n_39]|
      (And2 [a1, b2, n_39] |[a1]|
      (Xor2 [n_30, n_36, n_37] |[n_30, n_36]|
      (Xor2 [n_20, n_18, n_36] |[n_20, n_18]|
       (Or2 [n_34, n_31, n_35] |[n_34, n_31]|
       (Or2 [n_33, n_32, n_34] |[n_33, n_32]|
      (And2 [n_30, n_20, n_33] |[n_30, n_20]|
      (And2 [n_20, n_18, n_32] |[n_18]|
      (And2 [n_18, n_30, n_31] |[n_30]|
      (And2 [a2, b1, n_30] |[b1]|
      (Xor2 [n_22, n_28, output_0] |[n_22, n_28]|
      (Xor2 [n_21, n_19, n_28] |[n_21, n_19]|
      (Or2 [n_26, n_23, n_27] |[n_26, n_23]|
      (Or2 [n_25, n_24, n_26] |[n_25, n_24]|
      (And2 [n_22, n_21, n_25] |[n_22, n_21]|
      (And2 [n_21, n_19, n_24] |[n_19]|
     (And2 [n_19, n_22, n_23] |[n_22]|
      And2 [a1, b1, n_22]...
```

Conclusion

- LOTOS Code Generation from HDCaml
  - → The technique was examined about the procedure for generating the LOTOS code of N-bit from the circuit description in HDCaml.
  - → The LOTOS code generator module was extended that solved the problems of internal connected line, Bit operation, and process synchronization by mapping to DILL library.
  - → Finally N-bit structural specification of LOTOS code was successfully generated from High-Level HDCaml code.
- Behavioural Specification of the generated DILL library components is coded by hand to verify the generated LOTOS structural specification Code.

Future Tasks

- There is incompleteness for solving the problem of connecting wire in repeated structures of N-Bit DILL components, for instance MComp(Count, Connecting Wire, Component) produces multiple instances of the component process, so there must be some connection line in between the components processes which is not shown in this research.

- We also need some improvement for the automatic generation of the behaviour expression of the generated DILL library components for N-bit.

References

- ISO/IEC 8807:“Information Processing System, Open Systems Interconnection, LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour,” 1989
- T. Hawkins : HDCaml : http://www.confluent.org/wiki/doku.php/hdcaml.
- INRIA, France :Objective Caml : http://caml.inria.fr/
- J.He, K.J.Turner, “Extended DILL: Digital Logic in LOTOS,” Technical Report CSM-142, University of Stirling, 1999.
- Karl Flicker : HDCaml improvements :http://karl-flicker.at/hdcaml/
- C. A. R. Hoare: “Communicating sequential processes”, Communications of the ACM, Volume 21 , No.8 pp.666-677, 1978.

References

- CADP (Caesar/Aldebaran Development Package), A Software Engineering Toolbox for Protocols and Distributed Systems, INRIA/VASY, France, http://www.inrialpes.fr/vasy/cadp/
- ISO/IEC15437: “Enhancements to LOTOS (E-LOTOS),” 2001
- H. Ehrig, B. Mahr: “Fundamentals of Algebraic Specification, Part 1,” Springer Verlag, Berlin, 1985.

Thank You