SCS&E Report 9306
April, 1993

VHDL vs Functional Hardware Description: A Comparison and Critique

P. Kanthamanon, G. R. Hellestrand and M. C. Kam

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING
THE UNIVERSITY OF NEW SOUTH WALES

Abstract

This paper presents a comparison of two Hardware Description Languages (HDL)- VHDL and MODAL which employ different description styles for hardware specification. The comparison is both qualitative and quantitative and based on examples written in both languages. The languages are distinct in their power to describe hardware at various levels of abstraction. The results show that the functional description style, as used in MODAL, provides a more accurate description of hardware and modelling of hardware timing without loss of behavioural descriptive power.

1. Introduction

The objective of a Hardware Description Language (HDL) is to provide a suitable syntax and semantics for the formal specification of hardware. For behavioural descriptions, most HDLs employ sequential semantics, for example the VHSIC Hardware Description Language (VHDL). In addition to the distinct structural and behavioural modes of description, VHDL also provides a dataflow mode of description for more accurate hardware modelling.

Another style of hardware description is the functional style, as used in the Backus functional programming notation [BACK78]. One example of an HDL having this style is MODAL [HELL80], which is a concurrent, block structured, functional hardware design/description language satisfying the requirements of digital hardware description at the behavioural register–transfer, gate, and switch levels, using a common, simple and extensible notation.

It is difficult to assert whether a sequential or functional description style is most appropriate for behavioural specification. This paper compares both styles of description by comparing VHDL and MODAL specifications. Other features of both languages are discussed. The comparison is both qualitative and quantitative, and is based on examples of descriptions written in both languages. An analysis is presented of the three main applications of HDLs: specification, simulation and synthesis.

2. The Qualitative Comparison

2.1 Multi–level Hardware Description

A good HDL should provide a mechanism to describe hardware at all levels of abstraction from the behavioural to the structural domain. It should facilitate mixed–level descriptions and support both top–down and bottom–up design methodology thereby enabling designers to specify hardware from the architectural level to the structural level, and to model the real characteristics of hardware.

VHDL provides these features by employing separate and distinct structural, dataflow, and sequential notations, and supporting mixed descriptions as shown in Example 1, which describes an asynchronous data transfer unit with a 2–cycle signalling protocol. The block diagram of the asynchronous data transfer unit is shown in Figure 1. The primitive components used in structural descriptions have limited correspondence to real hardware. This will be discussed in the quantitative comparison.

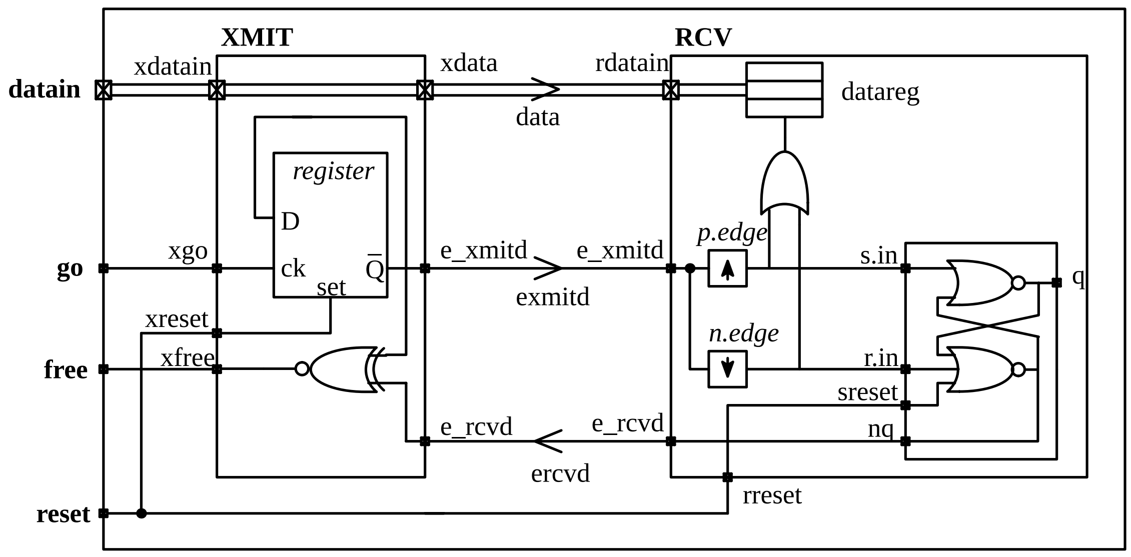


Figure 1. Block Diagram of an Asynchronous Data Transfer Unit with 2–Cycle Protocol

```
PACKAGE support IS
  TYPE bitvec IS ARRAY(3 DOWNTO 0) OF bit ;
  TYPE threeval IS ('0', '1', 'Z') ;
  TYPE threevec IS ARRAY (3 DOWNTO 0)
                  OF threeval ;
  FUNCTION bitvec2threevec(x : bitvec)
          RETURN threevec ;
  FUNCTION threevec2bitvec (x : threevec)
          RETURN bitvec ;
END support ;

USE WORK.support.ALL ;
ENTITY xmit IS
  PORT ( xgo, e_rcvd, xreset   : IN bit ;
         xdatain               : IN bitvec ;
         xfree, e_xmitd        : OUT bit ;
         xdata                 : OUT threevec );
END xmit ;
ARCHITECTURE behaviour OF xmit IS
  SIGNAL busy : bit;
BEGIN
```
— Sequential description
```
  PROCESS (xreset, xgo)
  BEGIN
    IF xreset = '1' THEN
      e_xmitd <= 0;
    ELSIF  NOT xgo'STABLE AND
          xgo = '1' THEN
      e_xmitd <= NOT e_xmitd ;
    END IF;
  END PROCESS ;
  PROCESS (xgo, busy)
  BEGIN
    IF xgo = '1' OR busy = '1' THEN
      xdata <= bitvec2threevec(xdatain) ;
    ELSE
      xdata <= "ZZZZ" ;
    END IF;
  END PROCESS ;
```
— Dataflow description
```
  busy <= e_xmitd XOR e_rcvd ;
  xfree <= NOT busy ;
END behaviour ;

USE WORK.support.ALL ;
ENTITY rcv IS
  PORT ( e_xmitd, rreset  : IN  bit ;
         rdatain          : IN threevec ;
         e_rcvd           : OUT bit );
END rcv ;
ARCHITECTURE behaviour OF rcv IS
  SIGNAL datareg : bitvec ;
BEGIN
```
— Sequential Description
```
  PROCESS (e_xmitd)
  BEGIN
    IF NOT e_xmitd'STABLE THEN
      datareg <= threevec2bitvec(rdatain);
    END IF;
  END PROCESS ;
  PROCESS (rreset, e_xmitd)
  BEGIN
    IF rreset = '1' THEN
      e_rcvd <= TRANSPORT '0' AFTER 100 ns;
    ELSIF NOT e_xmitd'STABLE THEN
      IF e_xmitd = '1' THEN
        e_rcvd <=  '1' AFTER 100 ns;
      ELSIF e_xmitd = '0' THEN
        e_rcvd <=  '0' AFTER 100 ns;
      END IF ;
    END IF;
  END PROCESS ;
END behaviour ;

USE WORK.support.ALL ;
ENTITY sync IS
  PORT ( reset, go   : IN bit  ;
         datain      : IN bitvec ;
         free        : OUT bit) ;
END sync ;
```
—Structural description
```
ARCHITECTURE structure OF sync IS
  COMPONENT xmit
    PORT ( xgo, e_rcvd, xreset : IN bit ;
           xdatain             : IN bitvec ;
           xfree, e_xmitd      : OUT bit ;
           xdata               : OUT threevec);
  END COMPONENT ;
  COMPONENT rcv
    PORT ( e_xmitd, rreset  : IN bit ;
           rdatain          : IN threevec ;
           e_rcvd           : OUT bit );
  END COMPONENT ;
  SIGNAL data    : threevec ;
  SIGNAL exmitd  : bit ;
  SIGNAL ercvd   : bit ;
BEGIN
  XM: xmit PORT MAP (go, ercvd, reset, datain,
                     free, exmitd, data) ;
  RC: rcv  PORT MAP (exmitd, reset, data,
                     ercvd) ;
END structure ;
```

Example 1. VHDL Descriptions of a Data Transfer Unit with an Asynchronous, 2–Cycle Protocol using Behavioural, Dataflow and Structural Descriptions.

While VHDL descriptions have limited correspondence to hardware, MODAL descriptions provide good correspondence by using a functional style of notation which employs functional forms. MODAL describes digital systems at all levels using a single, uniform notation as shown in Example 2, which describes the asynchronous data transfer unit in Figure 1. The semantics of each operation or function in MODAL correspond closely to actual hardware behaviour.

```
module sync (p.in reset, go, datain[3..0];
                p.out free)
  function /*function definition*/
    not ::    0 1     –> 1 0 ;
    and ::    1{1}  –> 1 0 ;
    or   ::    0{0}  –> 0 1 ;
  form /*form definition*/
    exnor ::    or[and, and not] ;
    exor   ::    not exnor ;
    nor     ::    not or ;
  module
    xmit (p.in xgo, e_rcvd, xreset, xdatain[3..0] ;
            p.out xfree, e_xmitd, xdata[3..0])
    register    xmit.state ;
    {
      xfree        <= exnor e_xmitd e_rcvd ;
      e_xmitd    <= not xmit.state ;
      ?or go exor e_xmitd e_rcvd?
                          xdata <= xdatain ;
      ?xreset?       xmit.state <– 1;
      /* ^ is signal rising detector*/
      ?and ^xgo not xreset?
                    xmit.state <– not xmit.state;
    }
  module rcv (p.in e_xmitd, rreset, rdatain[3..0];
            p.out e_rcvd)
  form
    p.edge      :: nor [not, D(25)];
    n.edge      :: nor [not D(25), a];
    any.edge   :: or [p.edge, n.edge] ;
  register    datareg[3..0];
  module srff[q, nq] (p.in s.in, r.in, sreset)
    {
      q     <= nor nq s.in ;
      nq   <= nor q sreset r.in;
    }
/*rcv: body*/
  {
    e_rcvd <= D(100) srff[nq](p.edge e_xmitd,
                            n.edge e_xmitd, rreset);
    ?any.edge e_xmitd? datareg <– rdatain ;
  }
net     data[3..0], exmitd, ercvd ;
/* start of machine SYNC */
{
  xmit(go, ercvd, reset, datain, free,
        exmitd, data);
  rcv(exmitd, reset, data, ercvd);
}
```

Example 2. The MODAL Descriptions of a Data Transfer Unit with an Asynchronous, 2–Cycle Protocol using Functional Style with Forms and Structural Style.

2.2 Behavioural Specification

In VHDL, a sequential style of description is used for behavioural specification. Many operators, such as Boolean functions, arithmetic operators, relational operators are provided. Such descriptions can only represent the intent of the hardware, the actual correspondence is weak. While this type of description is useful for indicating system function, it measurably complicates the synthesis process.

To provide high level descriptions, MODAL employs functional forms which are mathematically well–based, can describe the behaviour of hardware at a high level of abstraction, and can be readily synthesised and interpreted by a simulator.

Examples 3 and 4, below, show behavioural descriptions in VHDL and MODAL. Both descriptions represent the behaviour of a 16–bit serial adder. While the syntactic styles are very different, these two descriptions are behaviourally identical.

```
ENTITY sadd16 IS
   PORT ( a, b     : IN bit_vector(15 DOWNTO 0) ;
         load, clk  : IN bit ;
         sum        : OUT bit_vector(15 DOWNTO
0);
         cout       : OUT bit ) ;
END sadd16b ;
ARCHITECTURE sequent OF sadd16b IS
BEGIN
   PROCESS (load, clk)
      VARIABLE    aa, r   : bit_vector(15 DOWNTO
                                                 0);
      VARIABLE    count : integer ;
      VARIABLE    c        : bit ;
   BEGIN
      IF (NOT load'STABLE) AND (load = '0')
THEN
         aa       := a ;
         r         := b ;
         count := 0 ;
         c         := '0' ;
      END IF ;
      IF count /= 16 THEN
         IF NOT clk'STABLE THEN
            IF clk = '1' THEN
               r  := (aa(0) XOR r(0) XOR c)
                        & r(15 DOWNTO 1);
               c  := ((a(0) OR r(0)) AND c) OR (aa(0)
                        AND r(0));
            ELSIF clk = '0' THEN
               aa       := '0' & aa(15 DOWNTO 1);
               count := count + 1;
            END IF;
         END IF ;
      END IF ;
      sum    <= r ;
      cout   <= c ;
   END PROCESS ;
END sequent ;
```

Example 3. The Behavioural Description of VHDL

```
module sadd16(p.in a[15..0], b[15..0], c.in, load,
                                    clk;
                        p.out sum[15..0], cout)
   form  /*form definition*/
      equ    :: or[and,and not];
      exor   :: not equ;
      /*logical shift right*/
      shr     :: [0, a..w–1];
      /*incrementor*/
      inc     :: [exor[a, and[a+1..w, 1]], inc[a+1..w];
      /*1–bit full adder*/
      gen     :: and a..a+1;
      prop   :: exor a..a+1;
      sum    :: prop[prop, a+2];
      carry  :: or[gen, and[prop, a+2]];
   register /*register instantiation*/
      c, /*intermediate carry*/
      count[4..0],  /*counter*/
      r[15..0], /*result*/
      aa[15..0];  /*operand*/
   {
      sum cout <= r c ;
      ?!load? /*signal falling*/
      {
         aa       <– a ;
         r         <– b ;
         c         <– 0 ;
         count <– 0 ;
      }
      ?not count[4]?
      {
         ?^clk? /*signal rising*/
         {
            r     <– (sum a[0] r[0] c) r[15..1] ;
            c     <– carry a[0] r[0] c ;
         }
         ?!clk?
         {
            aa       <– shr aa ;
            count <– inc count ;
         }
      }
   }
```

Example 4. The Behavioural Description of MODAL

Since the default basis in MODAL is Boolean, the more complex operations such as multiplication and division may be defined by using forms or module descriptions. Recent extensions to MODAL [KAM92a] enable externally defined functions of arbitrary complexity to be incorporated into a MODAL description, with the same status as intrinsic functions. This strategy has obvious advantage in simulation where the high level function takes significantly less time to simulate than the equivalent function expressed in terms of primitive circuits.

Example 5 shows a MODAL description using high–level externally defined functions stored in a library. The function *u2f* in the example is used to convert data from unsigned binary, which is the default basis data type in MODAL, to floating point which is required by functions *fadd* and *fmul*.

```
#interface   <math.m>
/* The high–level operations library*/
            /* fadd  : floating point adder */
            /* fmul  : floating point multiplier */
            /* u2f   : unsign binary to foating point
                       converter */
module  SecondOrderFilter
       ( p.in ck, in[31..0]; p.out out[31..0])
register
  z[31..0];  zz[31..0];
net
  x[31..0];
constant
  k :: X’0001 ;
  g :: X’0001 ;
{
  ? ^clk? {
              zz  <– z;
              z   <– x;
           }
  /* x = in + z X k + z X g  */
  x    <= fadd in  fadd ( fmul z (u2f k))
           fmul z u2f g ;
  out <= zz;
}
```

Example 5. MODAL Description using Externally Defined Functions

2.3 Description Complexity

One important objective of HDLs is the formal specification of circuits. A description language should have well defined and consistent semantics and syntax. If the language allows designers to use different styles of description such as sequential and concurrent styles, it should explicitly separate the two sets of semantics to minimize the confusion for designers when writing or reading circuit descriptions.

```
ARCHITECTURE use_signal OF mux4 IS
SIGNAL  temp : integer ;
BEGIN
  PROCESS ( a, b, i0, i1, i2, i3 )
  BEGIN
    temp <= 0;
    IF a = ’1’ THEN
      temp <= temp + 1;
    END IF;
    IF b = ’1’ THEN
      temp <= temp + 2;
    END IF;
    CASE temp  IS
      WHEN 0 => q <= i0 ;
. . . . .
    END CASE ;
  END PROCESS;
END use_signal;
```

Example 6. VHDL Description Showing the Mixed–Usage of Signals and Variables in a Sequential Statement.

```
ARCHITECTURE use_variable OF mux4 IS
BEGIN
VARIABLE  temp : integer ;
  PROCESS ( a, b, i0, i1, i2, i3 )
  BEGIN
    temp := 0;
    IF a = ’1’ THEN
      temp := temp + 1;
    END IF;
    IF b = ’1’ THEN
      temp := temp + 2;
    END IF;
    CASE temp  IS
      WHEN 0 => q <= i0;
. . . . .
    END CASE ;
  END PROCESS;
END use_variable;
```

Example 7. The Correction for Example 6.

VHDL allows designers to have both signal assignments and variable assignments in sequential statements. Signal and variable assignments have different semantics, in that variables reflect their assigned values immediately whereas signals are updated after a time delay (possibly a process invocation delay). This is complex and confusing, and readily leads to unexpected behaviour in models.

Example 6 [PER91], above, attempts to directly and simply model the behaviour of a 4–to–1 multiplexer. The signal *temp* is initialised in the process statement (which are used for describing sequential behaviour). Since a signal object in VHDL does not get updated in the same simulation cycle as the value is assigned, the statements which rely on the value of *temp* are evaluated incorrectly. The error in Example 6 may be corrected by using VHDL variables, since they are updated as part of the assignment. Example 7 [PER91] shows the correct description of a 4–to–1 multiplexer.

In MODAL, all functions are assumed to perform work which requires energy and takes time, and is reflected as a definable delay which is consistently and straightforwardly interpretable by a simulator.

2.4 Technology Dependencies

Ideally, a description should be independent of implementation technology so that when technology changes, the same description should be still applicable. This enhances the portability of models written in the HDL.

Technology dependent information which is mainly used by tools such as simulators and synthesisers, should not be embedded in descriptions. Examples of technology dependent information are propagation delays, resolution functions and multiple–valued logic information.

```
ARCHITECTURE conflict OF simple IS
TYPE trivec IS ARRAY (integer RANGE <> )
                        OF threeval ;
FUNCTION resolve ( SIGNAL s : trivec )
    RETURN threeval IS
—Assume that both 1’s and 0’s predominate over Z
  VARIABLE  temp : threeval ;
BEGIN
  FOR i IN s’RANGE LOOP
    IF s(i) = ’0’ THEN
      temp := ’0’;
      exit ;
    ELSE
      temp := ’1’;
    END IF;
  END LOOP;
  RETURN temp;
END resolve ;
SIGNAL a : resolve threeval ;
BEGIN
  P1: PROCESS (in1)
      BEGIN
        IF in1 = ’1’ THEN
          a <= d1 ;
        ELSE
          a <= ’Z’ ;
        END IF ;
      END PROCESS P1 ;
  P2: PROCESS (in2)
      BEGIN
        IF in2 = ’1’ THEN
          a <= d2 ;
        ELSE
          a <= ’Z’ ;
        END IF ;
      END PROCESS P2 ;
  q <= a ;
END conflict ;
```

Example 8. VHDL Description using a Resolution Function

VHDL provides mechanisms for describing technology related information, which makes such descriptions technology specific and, as well, complicates both the syntax and semantics of the languages. Although VHDL provides library and package facilities which may be used to encapsulate technology dependent information, the language compiler must process this information so that it can be passed to various simulation and synthesis tools. If the description is to be used with different technology, a recompilation using the appropriate library and/or package is required. An example of a technology dependent VHDL description is shown in Example 8, above.

In this description, a conflict occurs as a result of more than two drivers assigning different values to the same signal. In VHDL, a resolution function is required to resolve the conflict. In this example, the resolution function is a Wired–AND connection. If TTL or CMOS technology is required, this resolution function correctly models the open–collector circuit. However, it is inappropriate for ECL technology because an open collector circuit in ECL performs a Wired–OR function.

In MODAL, the only abstraction which can be used to model technology dependent features is the delay operator. All technology dependent information has to be provided from other sources. For example, a simulator for MODAL has access to resolution functions and delay characteristics for various technologies. Example 9 illustrates a technology independent description in MODAL, in which there may be a conflict on the net *q* when both *in1* and *in2* are true. If the net *q* is driven by two drivers, the conflict will need to be resolved by the simulator, since it is part of the dynamic behaviour of the circuit.

```
module conflict ( p.in in1, in2, d1, d2 ; p.out q)
{
   ? in1 ? q <= d1;
   ? in2 ? q <= d2 ;
}
```

Example 9. MODAL Description Represent the Conflict of a Signal

```
module conflict ( p.in in1, in2, d1, d2 ;
                  p.out < TECH = TTL_OC> q)
/* use TTL_OC to resolve conflicts , if any */
{
   ? in1 ? q <= d1;
   ? in2 ? q <= d2 ;
}
```

Example 10. Using an Affix List to Pass Information to a Module in MODAL

Although MODAL does not provide constructs for the explicit specification of technology dependent information, the language provides a construct called an affix list [HELL80, KAM88] through which technology–dependent information may be described. This information is not interpreted by the compiler, but is passed directly to the simulators and synthesisers. Example 10 demonstrates the use of an affix list to specify technology dependent information.

3. The Quantitative Comparison

3.1 Accuracy in Modelling Timing and Delay

The method used to measure how accurately timing can be modelled by the two languages employs simulators to evaluate a set of examples written in each language. The results are compared with how real hardware is expected to perform. The VHDL toolset, PICA.VHDL Version 2.216 [PICA91], is used to simulate VHDL descriptions. Since there are many limitations in this toolset, certain techniques are used to facilitate the simulations. The MODAL compiler and Maxim simulator [KAM92b] are used to compile and simulate the MODAL descriptions. The results are interpreted according to the semantics of the simulation mechanism of each language, and the comparison covers different styles of description ranging from structural to behavioural.

In the structural domain, both languages employ component instantiation statements to describe the netlist of a circuit. Each component created is treated as a black–box whose functionality has to be described using some style of description. The accuracy in the interpretation of circuits described in the structural domain obviously relies on the accuracy of the other types of description employed in the individual component. The functionality of a component in VHDL can be described using dataflow or sequential descriptions, whereas only functional descriptions are supported in MODAL. Examples of various descriptions have been simulated to measure the accuracy of interpretation.

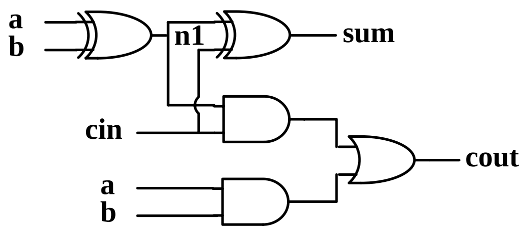


Figure 2. A Full Adder

```
ENTITY fadder IS
   PORT ( a       : IN bit ;
          b       : IN bit ;
          cin     : IN bit ;
          sum     : OUT bit :
          cout    : OUT bit );
END fadder ;
ARCHITECTURE dataflow OF fadder IS
   SIGNAL  n1 : bit ;
BEGIN
   n1     <= a XOR b AFTER 2 ns ;          — L1
   sum    <= n1 XOR cin AFTER 2 ns ;       — L2
   cout   <= (n1 AND cin) OR (a AND b)
                AFTER 2 ns;                — L3
END dataflow ;
```

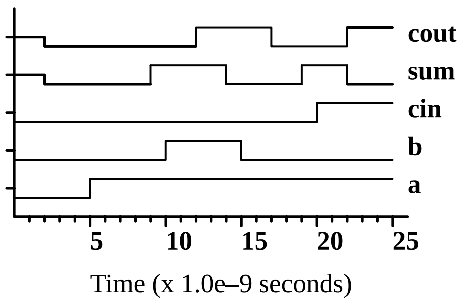
Example 11. VHDL Description of a Full Adder

```
module fadder (p.in a, b, cin ; p.out sum, cout)
net    n1;
{
   n1      <= exor a b ;
   sum     <= exor n1 cin ;
   cout    <= or (and a b) and cin n1 ;
}
```

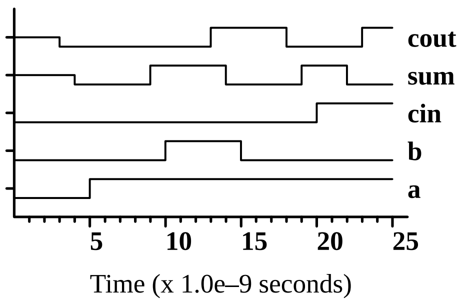
Example 12. MODAL Description of a Full Adder

```
module fadder (p.in a, b, cin ; p.out sum, cout)
net    n1;
{
   n1      <= D(5) exor a b ;
   sum     <= D(5) exor n1 cin ;
   cout    <= or (and a b) and cin n1 ;
}
```

Example 13 The use of Delay Operators in MODAL Description



(a) VHDL waveforms for a full adder



(b) MODAL waveforms for a full adder

Figure 3. Simulation Waveforms Produced by VHDL and MODAL Simulators

Figure 2 and Examples 11 and 12 describe a full adder, using the VHDL dataflow style and the MODAL functional style. Even though operator precedence rule and parentheses are used in VHDL, there is no delay property for operators embedded in the same statement. A lumped delay can be defined by using the *after* clause, as shown in Example 11. This causes on incorrect behaviour to be observed in the circuit associated with statement L3 in Example 11 (*cout* and *sum* should not change at the same time). To produce a correct description, delays have to be associated with each operator. The functional semantics of MODAL avoid this problem, since there is an intrinsic delay associated with the action of each operator and function. Designers can alter the desired delay for each signal by using the delay operator, as illustrated in Example 13. As a result, the MODAL description models the correct timing behaviour of the hardware. The simulation results for both VHDL and MODAL, as shown in Figure 3, confirm this comparative analysis.

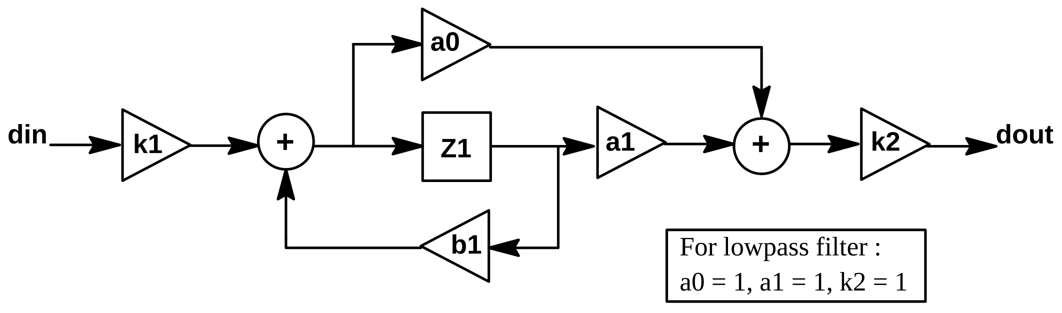


Figure 4. Block Diagram of a First–Order FIR Filter

```
PACKAGE support1 IS
   FUNCTION int2vec (x : integer)
         RETURN bit_vector(7 DOWNTO 0) ;
   FUNCTION vec2int
                        (x : bit_vector(7 DOWNTO 0))
         RETURN integer ;
END support1 ;

USE WORK.support1.ALL ;
ENTITY filter1 IS
   PORT (  din  : IN bit_vector(7 DOWNTO 0);
            ck   : IN bit;
         dout   : OUT bit_vector(7 DOWNTO 0));
END filter1;
ARCHITECTURE sequent OF filter1 IS
   SIGNAL z1d   : integer ;
   SIGNAL z1    : integer ;
   FUNCTION mult2c8(mand, mier : integer )
      RETURN integer IS
   VARIABLE ma     : integer ;
   VARIABLE ma1   : bit_vector(7 DOWNTO 0) ;
   VARIABLE md     : bit_vector(7 DOWNTO 0) ;
   VARIABLE mr     : bit_vector(7 DOWNTO 0) ;
   VARIABLE f       : bit ;
   VARIABLE i       : integer ;
```

Example 14. VHDL Description of a First–Order FIR Filter

```
module filter1 (p.in din[7..0], ck;
            p.out dout[7..0])
   #include <form.m>
   module adder[7..0](p.in a[7..0], b[7..0], cin)
      net  c[7..0] ;
      {
         adder <=  exor a : b : c[6..0] cin ;
               c  <=  or (and a : b) : and c[6..0] cin :
                           exor a : b ;
      }
   module mult2c8[mout[7..0], done]
            (p.in mand[7..0], mier[7..0], load)
      register
         ma[7..0]; md[7..0]; mr[7..0]; count[3..0];
      net     cki;
      initiate
         cki <= 1; count <– 0;
      {
         ? ^load ?
         {
            ma     <– 0;
            md     <– mand;
            mr     <– mier;
            count <– 1000 ;
         }
```

Example 15. MODAL Description of a First–Order FIR Filter

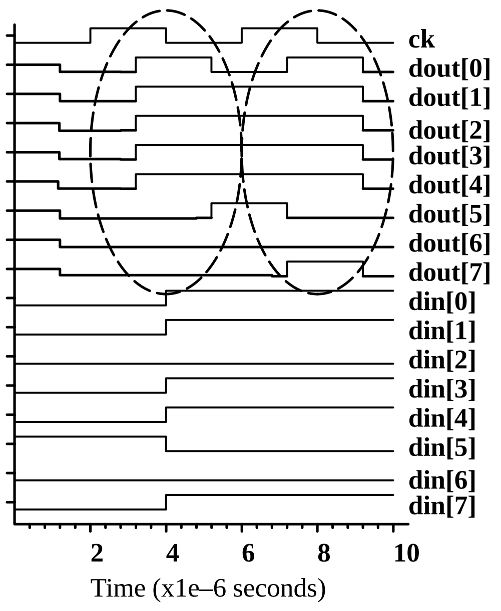
```
  BEGIN
    mr(7 DOWNTO 0)   := int2vec(mier) ;
    md(7 DOWNTO 0)   := int2vec(mand) ;
    f     := '0';
    ma  := 0 ;
    FOR i IN 0 TO 6 LOOP
      IF mr(0) = '1' THEN
        ma := ma + mand ;
      END IF;
      ma1    := int2vec(ma) ;
      f        := (md(15) AND mr(0)) OR f ;
      mr(7 DOWNTO 0)   := ma1(0) &
                            mr(7 DOWNTO 1);
      ma1(7 DOWNTO 0) := f &
                            ma1(7 DOWNTO 1);
      ma     := vec2int(ma1) ;
    END LOOP;
    IF mr(0) = '1' THEN
      ma := ma – mand;
    END IF;
    mr(0) := 0;
    RETURN ma ;
  END mult2c8;

BEGIN
  — Delay signal
  PROCESS (ck)
  BEGIN
    IF NOT ck'STABLE AND ck = '0' THEN
      z1d <= z1 ;
    END IF;
  END PROCESS;
  — Calculate output at tn
  PROCESS (ck, zid)
    VARIABLE m1, m2, m3, m4 : integer ;
    CONSTANT b1 : integer := –120 ;
    CONSTANT k1 : integer :=  124 ;
  BEGIN
    IF NOT ck'STABLE and ck = '1' THEN
      m1  := mult2c8(vec2int(din), k1) ;
      m2  := mult2c8(z1d, b1,) ;
    END IF ;
    m3 := m1 + m2 ;
    — Since a0, a1, k2 :=1, Calculate m4
    m4 := m3 + z1d ;
    — transfer new output
    z1                     <= m3 ;
    dout(7 DOWNTO 0)   <= int2vec(m4)
                               AFTER 1200 ns ;
  END PROCESS ;
END sequent ;
```

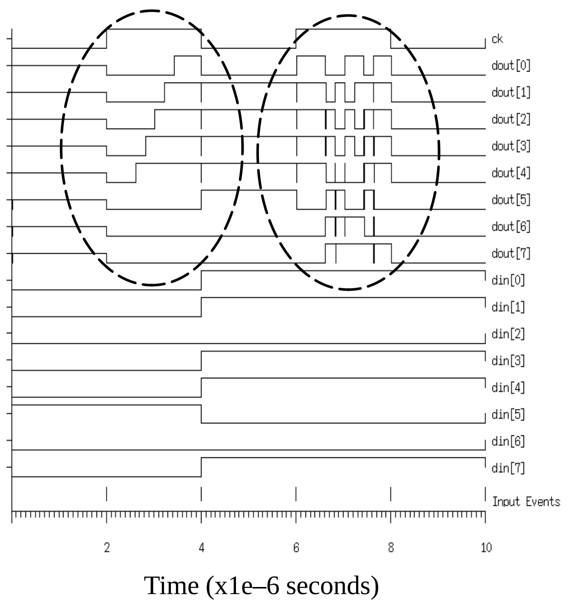
Example 14. VHDL FIR Filter (Cont.)

```
      ??
      {
        ? or count ?
        {
          ? ^cki ?
          {
            ?and (not or count[3..1]) count[0]?
            {
              ma     <– (?mr[0]?
                             adder(ma, not md, 1)
                         ?? ma);
              mr[0] <– 0;
            }
            ??
            {
              ma[7] <– or (and md[7] mr[0])
                           ma[7];
              ma[6..0] mr  <– (?mr[0]?
                                 adder(ma, md, 0)
                               ?? ma)
                               mr[7..1];
            }              count <– dec count;
          }
        }
      }
      mout  <= ma;
      done  <= not or count ;
      cki    <= D(100) not and cki or count ;
    }
register
  z1d[7..0];
net
  m1[7..0]; m2[7..0]; m3[7..0]; m4[7..0]; done1;
  done2;
constant
  k1 :: X'7c;
  b1 :: X'88;
initiate
  z1d <– 0;
{
  ? !ck ? z1d <– m3 ;
  m1 done1 <= mult2c8(din, k1, ck);
  m2 done2 <= mult2c8(z1d, b1, ck);
  m3          <= adder(m1, m2, 0);
  m4          <= adder(z1d, m3, 0);
  dout        <=  m4;
}
```

Example 15. MODAL FIR Filter (Cont.)



(a) Waveforms from VHDL simulation



(b) Waveforms from MODAL simulation

Figure 5. Simulation Waveforms of a First–Order FIR Filter Described in VHDL and MODAL

In the next step of this comparison, a first–order FIR filter is described in both languages to compare the VHDL sequential style and the MODAL functional style of behavioural description. The block diagram of the filter is shown in Figure 4, and the multiplication algorithm used is from [CAV84]. Examples 14 and 15 detail the sequential VHDL description and the functional MODAL description of the first–order FIR filter, respectively. The assignment statements of VHDL description support only lumped delay, and may cause the problems mentioned in Section 2.3. In contrast, the functional description in MODAL describes the functions needed for the first–order FIR filter, and still corresponds closely to the hardware behaviour. The output delay is evaluated according to which input change has occurred, and the number of levels of function delay incurred to propagate this signal change to the output. As a result, the timing delay as expected in actual hardware is obtained. Figure 5 shows the simulation results obtained from each CAD system.

3.2 Accuracy in Describing Hardware

Digital hardware is usually described in terms of modules and interconnections. The definition within modules details the functions and the hierarchical structure, while the interconnections specify the communication paths among modules. This information is important in synthesis from HDL specifications, since it facilitates the generation of data–flow and control–flow graphs.

Both VHDL and MODAL provide accurate hierarchical and interconnection information using structural description. The functions of modules are described differently, using the dataflow and sequential styles in VHDL, and the functional style in MODAL. The MODAL functional style provides good correspondence with hardware, which facilitates the synthesis task. The VHDL dataflow style has similar correspondence, but the sequential style provides little information for hardware synthesis, resulting in VHDL constructs being restricted to a synthesisable subset of the language [CAM91]. Since VHDL dataflow and sequential styles have different semantics, different techniques are required to synthesis each description style, complicating the task of synthesisers. Examples 11 and 14, and 12 and 15 together with Figures 2 and 4, illustrate the hardware correspondence issues described in this section.

4. Conclusions

The qualitative comparison demonstrates the similarity between VHDL and MODAL in their respective ability's to describe hardware at various levels of abstraction. At the structural level, circuit descriptions written in each language are similar in style and semantics. However, large differences in the description styles are evident at the behavioural level. While VHDL provides distinct dataflow and sequential styles of description at this level, MODAL uses a uniform functional notation.

VHDL allows designers to mix objects, namely signals and variables, having different delay semantics in the same sequential statements. This makes the language confusing and may lead to unexpected behaviour in the system being designed. MODAL, by contrast, provides consistent delay semantics.

The ability to describe technology dependent information in VHDL demonstrates a nexus between the roles of description/specification and interpretation, which reduces the reusability of the descriptions. The quantitative analysis reveals that the functional description used in MODAL provides a more accurate language for describing hardware, and modeling of timing and delay. This allows simulation tools to provide accurate result and eases the task of synthesisers.

In conclusion, the comparisons show that the functional description style of MODAL has a consistent concurrent semantics whereas VHDL has an mix of sequential and concurrent semantics. The many features of VHDL to support the many description styles increase the language complexity. The functional description style provides a more natural/intuitive semantics for hardware descriptions. This facilitates the simulation and automatic synthesis of hardware from MODAL descriptions.

References:

[ARM89] J.R. Armstrong, ”Chip–Level Modeling with VHDL”, Prentice Hall,, New Jersey, USA, 1989.

[BACK78] J. Backus,”Can programming be liberated from the von Neumann style: a functional style and its algebra of programs”,Communications of ACM, Vol. 21, No. 8, pp. 613–641, 1978.

[CAM91] R. Camposano, L.F. Saunders, and R.M. Tabet, ” VHDL as Input for High–Level Synthesis”, IEEE Design &Test of Computers, Vol. 8, No. 1, pp. 43–49, March 1991.

[CAV84] J. J. F., Cavanagh, ”Digital Computer Arithmetic, New york, 1984.

[HELL80] G.R. Hellestrand, ”MODAL: A System for Digital Hardware Description and Simulation”, Journal of Digital Systems, Vol. 4, No. 3, pp. 241–304, Fall 1980.

[IEEE87] ”IEEE Standard VHDL Language Reference Manual”, IEEE Std. 1076–1987, IEEE Computer Soc. Press, USA., 1987.

[KAM88] Ming Chi Kam, ”Tutorial on Using the MODAL Compiler and Simulation System”, Internal Document, UNSW., May 1988.

[KAM92a] M.C. Kam, ”Using the MODAL Math Functions”, Internal Document, VaST Lab., UNSW., 1992.

[KAM92b] M.C. Kam, ”MODAL – A Hardware Description Language: User’ Manual”, UNSW., 1992.

[PER91] Douglas L. Perry, ”VHDL”: Int. Ed., McGRAW–HILL, Singapore, 1991.

[PICA91] ”VHDL Toolsets Reference Manual”, PICA Lab., Uni. of Pittsburgh, USA., 1990.