

# Porting an RTL Synthesis Paradigm to VHDL

Mehran M. Massoumi  
Viewlogic Systems, Inc.  
2350 Mission College Blvd. Suite 1000  
Santa Clara, California 95054

Fredrick J. Hill  
Electrical and Computer Engineering Department  
University of Arizona  
Tucson, Arizona 85721

## Abstract

Lack of close hardware correspondence in the existing VHDL synthesis subsets results in circuits that are not minimal in terms of gate count and path delays. However, VHDL being a broad language, contains a number of constructs intended for synthesis which are not typically used in synthesis tools today. In this paper, we attempt to narrow the gap between VHDL and hardware by porting the constructs of a Register Transfer synthesis language to VHDL. AHPL (A Hardware Programming Language) is selected for this purpose. In addition to being one of the languages considered in designing VHDL, AHPL has stood the test of time in being a robust synthesis language. Synthesis from the resulting VHDL subset, referred to as  $VHDL^{RT}$ , is compared to other methodologies. Moreover, the pitfalls present in the existing subsets are emphasized. Subsequently, the features of AHPL as well as the mapping of AHPL into VHDL are presented.

## 1 Introduction

The emergence of VHDL [a,b] as an industry standard Hardware Description Language triggered many research efforts in the area of high level synthesis. Since VHDL possesses constructs that lack any clear hardware correspondence many companies [c] as well as universities [h] channeled resources into defining "synthesizable" subsets and/or styles in VHDL. Tools are developed that produce hardware in a target technology from these subsets. Early experience in synthesis suggests that lack of close hardware correspondence in the existing VHDL subsets results in circuits that are not minimal in terms of gate count and path delays. However, VHDL being a broad language, contains a number of constructs intended for synthesis which are not typically used in synthesis tools today. Therefore, in an attempt to narrow the gap between VHDL and hardware, we map the constructs of a

Register Transfer Synthesis language to those of VHDL. AHPL (A Hardware Programming Language) is the RTL language selected for this purpose. This mapping produces a synthesizable VHDL subset (referred to as  $VHDL^{RT}$  henceforth) which is as close to hardware as AHPL and yet possesses the convenience of description present in RTL languages. AHPL is a hardware description language based on the notational conventions of APL. Only those APL operations which can be readily interpreted as hardware primitives are used in AHPL. Since its introduction, AHPL has demonstrated to be a robust synthesis language and an effective tool in classroom environment [d,e].

Once we have defined the  $VHDL^{RT}$ , we will compare the synthesis from  $VHDL^{RT}$  to other methodologies. Although actually a slight superset, AHPL advocates may wish to consider  $VHDL^{RT}$  as the definitive mapping of AHPL into VHDL.

In Section 2 we highlight the pitfalls present in existing VHDL subsets. The features of AHPL are presented in section 3 and the mapping of AHPL into VHDL in section 4. The last 2 sections will include experimental results and concluding comments.

## 2 VHDL And Synthesis

The just emerging VHDL synthesis tools make more use of behavioral statements in VHDL for synthesis than data flow statements. It is the purpose of this section to analyze the problems associated with such approach. Pitfalls which fall into the following categories are introduced in a series of examples. (1) Don't cares are ignored. (2) Lack of correspondence between behavioral description and generated hardware.

To illustrate the problem in specifying the don't cares, we first present a simple case and then proceed to a more complex and realistic example. The following AHPL statement represents a bus connection. The values A, B, and C are placed on RBUS depending on the truth of the conditions c1, c2, and c3.

$$RBUS = (A!B!C) * (c1, c2, c3);$$

The bus conditions on the right hand side of the asterisk are assumed to be mutually exclusive and therefore the logic that is generated is as follows. ('&' is used for And operator and '+' for Or operator)

$$RBUS = (A\&c1) + (B\&c2) + (C\&c3)$$

A process is typically used in most existing VHDL subsets to describe the above hardware. Figure-1 illustrates such process description.

The syntax of VHDL does not allow specification or assumption of mutual exclusivity of c1, c2, and c3. As a result the following logic is generated:

$$RBUS = (A\&c1) + (B\&c2\&\bar{c1}) + (C\&c3\&\bar{c2}\&\bar{c1})$$

If the VHDL process of figure-1 is changed to disjoint if statements instead of one if statement with elsif's, one can see that different logic will result. Whether or not the second circuit is more optimized than the first is not under designer's control.

In the above example the penalty in terms of hardware may not seem significant but the example of figure-2 illustrates a case where the extra hardware can be a problem. The

```

process(A, B, C, c1, c2, c3)
begin
  if (c1 = '1') then
    RBUS <= A;
  elsif (c2 = '1') then
    RBUS <= B;
  elsif (c3 = '1') then
    RBUS <= C;
  end if;
end process;

```

Figure 1: Process Description

```

abus = (md!m̄d)*
  ((ir[0]&ir[1]&(ir[2] + ir[3]), (ir[0]&ir[1]&(ir[2] + ir[3])));
bbus = ac;
cin = (cf&ir[3]) + (ir[1]&ir[3]);
obus =
  (add[1 : 32](abus; bbus; cin)!abus&bbus!abus + bbus!abus@bbus!abus)*
  ((ir[0]&ir[2] + ir[0]&ir[3]), ir[0]&ir[3], ir[0]&ir[2]&ir[3],
  ir[0]&ir[2]&ir[3], ir[0]&ir[2]&ir[3]);
ac * ((ir[2]&ir[3]) <= obus;
cff * (ir[0]&ir[2] + ir[3]) <= add[0](abus; bbus; cin);
zff <= (+/obus);
nff <= obus[0];
vff * (ir[0]&ir[2] + ir[3]) <= (abus[0]&bbus[0]&add[1](abus; bbus; cin)+
  abus[0]&bbus[0]&add[1](abus; bbus; cin));

=> (1).

```

Figure 2: AHPL Description of an ALU

AHPL step shown is the ALU description of a 32-bit processor [d]. The statements in this figure represent the logical and arithmetic actions taken in a particular state of a processor. Based on the contents of the Instruction Register (ir), the control may reach the above state. As shown, The first four statements are connections to various buses which depending on IR, assume different values or computations. The last five statements represent transfer of information into registers and flags. The asterisk on the right hand side of a connection represents a bus. These conditions are symmetric in that order of appearance does not affect the generated hardware. It is easily verified that the obus conditions are not logically mutually exclusive. However, don't care combinations guarantee that no two conditions can be true at the same time. Control will never reach this step for instruction corresponding to these don't care conditions. The symmetry of the bus condition syntax in AHPL gives the designer full control over what will actually be the bus condition and he/she can therefore, take advantage of the don't care cases.

In VHDL, the obus expression is usually described in the manner shown in figure-3. Many

```

process( ... )
begin
    if (step = s22) then
        if ((ir(0) and ir(2) or not ir(0) and ir(3)) = '1') then
            obus <= addsig;
        elsif ((ir(0) and ir(3)) = '1') then
            obus <= abus and bbus;
        elsif ((ir(0) and not ir(2) and not ir(3)) = '1') then
            obus <= abus or bbus;
        elsif ((ir(0) and ir(2) and not ir(3)) = '1') then
            obus <= abus xor bbus;
        elsif ((not ir(0) and ir(2) and not ir(3)) = '1') then
            obus <= abus;
        else
            obus <= "0 ... 0";
        end if;
    end if;
end process;
end try;

```

Figure 3: Partial VHDL Description of an ALU.

tools [c] support this style. Typically, synthesis systems translate VHDL descriptions into boolean expressions of the form shown below and then a logic optimizer is used for minimization.

$$\begin{aligned}
 f1 &= ir[0] \& ir[2] + \overline{ir[0]} \& ir[3] \\
 f2 &= ir[0] \& ir[3] \\
 f3 &= ir[0] \& \overline{ir[2]} \& \overline{ir[3]} \\
 f4 &= \overline{ir[0]} \& ir[2] \& \overline{ir[3]} \\
 f5 &= \overline{ir[0]} \& \overline{ir[2]} \& ir[3] \\
 obus[i] &= compare(step, "10110") \& \\
 & ((addsig[i] \& f1) + \\
 & (abus[i] \& bbus[i] \& \overline{f1} \& f2) + \\
 & ((abus[i] + bbus[i]) \& \overline{f1} \& \overline{f2} \& f3) + \\
 & ((abus[i] @ bbus[i]) \& \overline{f1} \& \overline{f2} \& \overline{f3} \& f4) + \\
 & (abus[i] \& \overline{f1} \& \overline{f2} \& \overline{f3} \& \overline{f4} \& f5))
 \end{aligned}$$

The knowledge that f1, f2, f3, f4, and f5 are mutually exclusive can significantly reduce the logic for each element of obus. No optimizer can reduce this logic to its possible minimum without the don't care information.

In the case where the mutual exclusivity of f1 thru f5 was logically derivable, theoretically an optimizer can produce a minimal or close to minimal implementation. However, in such scenarios, the computation requirement can be prohibitive.

The point to be made is that if the VHDL subset is not well defined, the optimum circuit is not likely to result from synthesis. From a hardware designer point of view, a well defined synthesis subset is one which maintains a close hardware correspondence. In such languages, the designer knows how a change in the description affects the hardware.

### 3 Features of AHPL

The defining features of AHPL as a Language to support synthesis are (1) Its natural partition of a digital system into control and data section, and (2) The precise hardware correspondence of each AHPL primitive. The control circuit will cause register transfers to take place in the data section. In some systems the sequencing of control will be influenced by the branching information from the data section. Usually the memory elements in the data section are arranged as registers. A signal from the control unit will typically cause the results of a logical computation to be transferred into all flip flops of one or more registers. Since the bits of registers are often treated uniformly, these logical computations can be conveniently expressed in the vector notation of APL.

A complete AHPL description of a system consists of Combinational Logic Units (CLUnits henceforth) and modules. Module descriptions consist of a sequence of steps specifying data transfers and/or connections. Moreover, AHPL uses structured syntax to facilitate description of CLUnits. This makes it possible for a hardware compiler program to process a description step by step and to generate a network wire list for the corresponding CLUnit. Due to the one to one correspondence between AHPL constructs and hardware primitives, derivation of hardware from the description is a natural process.

### 4 Definition of Mapping to $VHDL^{RT}$

Identification of VHDL constructs intended for synthesis is the primary focus of this section. As mentioned above, an AHPL description consists of Module Descriptions and CLUnits. A module in AHPL consists of (1) Declarations, (2) Body of control state conditioned statements, and (3) Always active statements. An example-driven approach will be used to define the mapping of AHPL constructs into those of VHDL. The AHPL module in figure-4 describes a 2-of-5 code detector which has 4 inputs and 1 output. Once a one to zero transition on input 'cont1' is detected then input 'data' is sampled for a valid 2-of-5 code while 'cont1' is zero. The only output 'Z' will be one concurrent with the 5th 'data' bit if an invalid code is detected. A 3-state FSM and some datapath elements are the major ingredients of this circuit.

Shown in figure-5 is the mapping of input and output ports into VHDL. The VHDL keywords IN, OUT, and BUS are used to determine the direction and nature of the signal. The memory and bus elements are declared inside the architecture part of the VHDL description (figure-6). The VHDL keyword REGISTER is used to declare a signal as register. All carriers in  $VHDL^{RT}$  are declared as bus, register, or wire and there is no ambiguity as to their hardware implementation. The body of an AHPL description consists of a number of steps in which connections and/or transfers take place. As mentioned above, AHPL enforces separation of control and data. The same philosophy is ported to VHDL.

```

MODULE: CODE.
EXINPUTS  : DATA; CONTL; CLOCK; RESET.
EXOUTPUTS  : Z.
MEMORY    : ONES[2]; BITS[3].
CLUNITS: INC1[2] <: INCR{2}.
CLUNITS: INC2[3] <: INCR{3}.
BODY SEQUENCE:CLOCK.
1  ONES <= \0,0\;
   BITS <= \0,0,0\;
   => (^CONTL) / (1).

2  BITS * ^CONTL <= INC2(BITS);
   ONES * (^CONTL & DATA) <= INC1(ONES);
   => (CONTL) / (2).

3  BITS <= (INC2(BITS) ! 3$0) * (^CONTL, CONTL);
   ONES * (DATA + CONTL) <=
       (INC1(ONES) ! 2$0) * (^CONTL, CONTL);
   Z = ^CONTL & BITS[0] & ^((ONES[0] & ^ONES[1] & ^DATA) +
       (^ONES[0] & ONES[1] & DATA));
   => ((BITS[0] & ^CONTL), (^BITS[0] & ^CONTL), CONTL) /
       (1, 3, 2).

ENDSEQUENCE
CONTROLRESET(RESET)/(1).
END.

```

Figure 4: AHPL Description of 2-of-5 Code Detector.

AHPL Description	VHDL Description
MODULE: CODE.	ENTITY CODE IS
	PORT (
EXINPUTS: DATA;	>> >> DATA : IN RBIT; -- Resolved Bit
CONTL;	>> >> CONTL: IN RBIT;
CLOCK;	>> >> CLOCK: IN RBIT;
RESET.	>> >> RESET: IN RBIT;
EXOUTPUTS: Z.	>> >> Z : OUT RBIT BUS);
	END CODE;

Figure 5: Definition of AHPL - VHDL Mapping.

AHPL Description	VHDL Description
MODULE: CODE. ....	ENTITY CODE IS ....
....	END CODE; ARCHITECTURE ONE OF CODE IS
MEMORY: ONES[2]; BITS[3].	SIGNAL ONES: RBIT_VECTOR(0 TO 1) REGISTER; SIGNAL BITS: RBIT_VECTOR(0 TO 2) REGISTER;
BUSES: RBUS. BODY SEQUENCE: CLOCK.	SIGNAL RBUS: RBIT BUS; BEGIN
< <i>ModuleBody</i> > < <i>EndSequence</i> >	< <i>ArchitectureBody</i> >
END.	END ONE;

Figure 6: Definition of AHPL - VHDL Mapping.

Extracting the FSM from the AHPL code of figure-4 results in the following partial description.

```

1  => (~CONTL, CONTL) / (1,2).
2  => (CONTL, ~CONTL1) / (2,3).
3  => ((BITS[0] & ~CONTL), (~BITS[0] & ~CONTL), CONTL)/(1,3,2).

```

Every control step translates into a block statement with label con"number", where "number" is an integer representing the step number. The description of step 1 is depicted in figure-7.

All state variables are to be declared as an enumerated type and of kind register. The block guards are the conditions under which state transitions take place. Notice that the label for step 1 is CON1. In blocks labeled CON'number', only assignments to signals of type enumerated type is allowed. This allows a flexible state assignment rather than the one hot state assignment implied by the AHPL description.

The data section is mapped in a similar way with the difference that the bus control and the clock control are distinguished. Consider the following AHPL statement.

$$ONES * (COND1 + COND2) \leq (A!B) * (\overline{CONTL}, CONTL);$$

the expression COND1+COND2 is the clock condition whereas the expressions after the asterisk on the right hand side are the bus conditions. In  $VHDL^{RT}$ , we use the guard conditions for clock condition and conditional WHEN statments for bus conditions. This distinction is not made in the control part of  $VHDL^{RT}$  because the impact of such separation on the amount of control hardware is not significant. The  $VHDL^{RT}$  description of step 3 of the 2-of-5 Code Detector is shown in figure-8.

Lastly, a Combinational Logic Description in AHPL consists of (1) Declarations and (2) Body of connection statements. The Declaration consists of input list, output list, connection points, and list of nested clunits. The body of the Combinational Logic Unit contains the connection activities. The example of figure-9 which is the description of a generic increment logic illustrates the correspondence between AHPL and VHDL.

```

ARCHITECTURE ONE OF CODE IS
  TYPE    ENUM_STEP IS (S1, S2, S3);
  SUBTYPE STEP IS ONE_OF ENUM_STEP;    --Resolved
  SIGNAL MY_FSM: STEP:= S1 REGISTER;
BEGIN
  CON1: BLOCK ((MY_FSM = S1) AND (NOT CLK'STABLE) AND (CLK = '0'))
  BEGIN
    TMP1: BLOCK (GUARD AND ((NOT CONTL) = '1'))
    BEGIN
      MY_FSM <= GUARDED S1;
    END BLOCK TMP1;

    TMP2: BLOCK (GUARD AND (CONTL = '1'))
    BEGIN
      MY_FSM <= GUARDED S2;
    END BLOCK TMP2;
  END BLOCK CON1;
END ONE;

```

Figure 7: Partial *VHDL<sup>RT</sup>* Description of step 1 of 2-of-5 Code Detector.

```

ST3: BLOCK (MY_FSM = S3)
BEGIN
  B: BLOCK (GUARD AND NOT CLK'STABLE AND (CLK = '0'))
  BEGIN
    BITS <= GUARDED INC2 WHEN ((NOT CONTL) = '1') ELSE "000";
    B1: BLOCK (GUARD AND ((DATA OR CONTL) = '1'))
    BEGIN
      ONES <= GUARDED INC1 WHEN ((NOT CONTL) = '1') ELSE "00";
    END BLOCK B1;
  END BLOCK B;
  Z <= GUARDED (NOT CONTL) AND BITS(0) AND
    (NOT ((ONES(0) AND (NOT ONES(1)) AND (NOT DATA)) OR
      ((NOT ONES(0)) AND ONES(1) AND DATA)));
END BLOCK ST3;

```

Figure 8: Partial *VHDL<sup>RT</sup>* Description of step 3 of 2-of-5 Code Detector.

AHPL Description	VHDL Description
CLU:INCR(X)N.	ENTITY INCR IS GENERIC (N: INTEGER);
INPUTS:XIN[N]. OUTPUTS:XOUT[N].	PORT(XIN : IN RBIT_VECTOR(0 TO N-1); XOUT: OUT RBIT_VECTOR(0 TO N-1));
CTERMS:CUM[N].	END INCR;
BODY	ARCHITECTURE STRUCT OF INCR IS SIGNAL CUM: RBIT_VECTOR(0 TO N-1);
	BEGIN
XOUT[N-1] = XIN[N-1];	XOUT(N-1) <= NOT XIN(N-1);
CUM[N-1] = XIN[N-1]; FOR I = N-2 TO 0 CONSTRUCT	CUM(N-1) <= XIN(N-1); L1: FOR I IN N-2 DOWNT0 0 GENERATE
XOUT[I] = XIN[I]^CUM[I+1]; CUM[I] = XIN[I]&CUM[I+1]	XOUT(I) <= XIN(I) XOR CUM(I+1); CUM(I) <= XIN(I) AND CUM(I+1);
ROF.	END GENERATE L1;
END.	END STRUCT;

Figure 9: AHPL - VHDL Mapping of CLUnits

circuit	$VHDL_{NotOptimized}^{syn}$	$VHDL_{Optimized}^{syn}$	$VHDL^{RT}$
16-bit Signed Multiplier	725	574	369
ALU OBUS expression (1)	506	284	269
ALU OBUS expression (2)	506	386	272
2-of-5 Code Detector	88	57	56
11100 Sequence Detector	58	29	34

Table 1: Experimental Results (Gate Count).

## 5 The Role of Optimization

In this section we consider several examples in both  $VHDL^{RT}$  and a VHDL subset supported by most existing tools. The latter will be referred to as  $VHDL^{syn}$ .

An MIS [g] based logic synthesis tool was used to synthesize the  $VHDL^{syn}$  description of all circuits into a set of generic gates. Two synthesis runs were done for each example; one with area optimization turned on and the other with no optimization. Also, the  $VHDL^{RT}$  was synthesized into the same set of generic gates using the AHPL multi-stage compiler [i]. Very limited optimization is performed by the multi-stage compiler. Table-1 presents the gate count resulting from both synthesis systems for five test cases. The ALU OBUS expression (1) refers to the description of figure-3. The next row of table-1 refers to almost the same description with conditions f1 through f5 changed such that their mutual exclusivity is not logically derivable. It can be seen that even a powerful optimizer is of little help if there are pitfalls in the VHDL subset.

## 6 Conclusions

There are many constructs in VHDL language that are ignored by most existing synthesis subsets. Among these are the Register Transfer constructs that are essentially meant to have close correspondence to hardware. This paper is an attempt to highlight such Register

Transfer constructs and demonstrate their advantages in synthesis. The point to be made is that if the circuit behavior is described in a well defined VHDL subset then there is essentially little need for computationally expensive optimization. Moreover, in many cases having a powerful optimizer, such as the one in MIS tools, will be of little help if the original description is convoluted.

## 7 References

- [a] "IEEE Standard VHDL Language Reference Manual",  
IEEE Std 1076-1987, IEEE Inc., 1988
- [b] Lipsett, Schaefer, Ussery, "VHDL: Hardware Description  
and Design", Kluwer Academic Publishing, Boston, 1988.
- [c] Carlson, S, "Introduction to HDL-Based Design Using VHDL",  
Synopsis Inc.
- [d] Hill, F.J., Peterson, G.R. "Digital Systems: Hardware Organization  
and Design", 3rd Edition, John Wiley and Sons, New York, 1987.
- [e] Hill, F.J., et al, "Hardware Compilation from an RTL to a  
storage Logic Array Target", IEEE Trans. on Computers  
Aided Design. vol. CAD-3 pp208-217, July 1984.
- [f] Hill, F.J., "Introducing AHPL", IEEE Computer, Dec. 1974.
- [g] Brayton, Rudell, Sangiovanni-Vincentelli, Wang, "MIS: A  
Multiple-Level Logic Optimization System",  
IEEE Transactions on CAD, November 1987.
- [h] Lis, J., Gajski, D., "VHDL Synthesis Using Structured  
Modelling", 26th ACM/IEEE DAC, 1989.
- [i] Swanson, R. E. et al, "An AHPL Compiler/Simulator System",  
Proceedings Sixth Texas Conference on Computing Systems, Nov. 1977.