Modeling with SpecCharts

Sanjiv Narayan
Frank Vahid

Technical Report #90-20
July 25, 1990

Dept. of Information and Computer Science
University of California, Irvine
Irvine, CA 92717
(714) 856-7063

narayan@ics.uci.edu
vahid@ics.uci.edu

Abstract

SpecCharts is a language intended for system level description and synthesis. It is based on hierarchical state diagrams, posseses many constructs designed to facilitate ease of system level descriptions, and is simulatable via a translator from SpecCharts to VHDL. To test the feasability of using the language, several examples were modeled using SpecCharts, were converted to VHDL, and simulated to verify correctness. The details of two of those examples are provided in this report.

Contents

List of Figures

List of Tables

1 Modeling Controlled Counter using SpecCharts

This section describes the SpecChart representation of the Controlled Counter [Arms89]. Though SpecCharts are intended to specify complex systems at an abstract level, this section demonstrates the language's ability to model at a somewhat detailed level.

1.1 Controlled Counter Description

The Controlled Counter can count up or down on each rising clock, to a specified limit, and can be asynchronously cleared. It can be thought of as consisting of 3 main components (see figure 1). CONVAL stores the CON value and outputs its decoded signal indicating what the controlled counter should do next. LIM is a register whose value is used by the counter to determine when to stop counting (when the limit has been reached). COUNTER is the component that actually contains the current count value and performs the incrementing or decrementing based on CONVAL, LIM, and the clock input, and outputs this register as the controlled counter's count output.

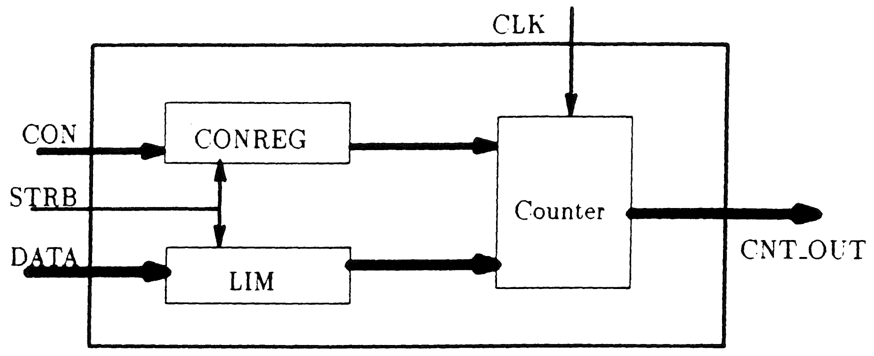


Figure 1: Block diagram of the Controlled Counter

When the STRB (strobe) input is rising, CON is stored. The controlled counter will then perform one of the following actions based on the stored value of CON:

- "00" : Clear (current count becomes 0)
- "01" : Load the limit with the value on the DATA input line on the falling edge of STRB
- "10" : Count up on rising clock unless limit is reached
- "11" : Count down on rising clock unless limit is reached

Note that any sequence and combination of count ups, count downs, limit loads, and clears are allowed, permitting rather strange sequences such as loading a limit of 7, counting from 0 to 3, changing the limit to 1, and then counting from 3 on up. An alternative design might require that the limit be reached before a new limit can be loaded; this might capture the true intention of usage and would result in a much simpler specification. However this design is not considered here as our purpose is to model the same design as in [Arms89].

Several timing details of the counter's operation are included in the SpecChart. See [Arms89, LiGa89] for a description of those details and for another description of the controlled counter's operation.

1.2 The SpecChart Model

Figure 2 shows the SpecChart specification of the controlled counter. The description above gave three main functions that must be performed: loading and decoding CON, loading LIM, and counting. There are two other functions needed: updating the output ports to reflect the internal count value, and generating an internal enable signal which disables counting if the limit has been reached. This implies the controlled counter can be modeled by five concurrent substates, one for each function. Each of the five states is now briefly discussed.

- **Decode**– Loads the condition register CONREG with CON, then generates a decoded signal CONSIG. It will do this whenever STRB is rising.
- **Load_limit**– Loads the limit register LIM when CONSIG is "0010" AND STRB is falling. The wait statement used is *wait on STRB until STRB='0' and CONSIG(1)='1'*. There is a VHDL intricacy to note here. The *on STRB* can not be ommitted, since then by default VHDL would add on STRB,CONSIG, which is incorrect, since a change on CONSIG can not trigger the loading of LIM.
- **Update_enable**– Generates an enable signal EN which disables counting if the current count value equals the limit. The code waits on a change of the count or the limit, which are the only two things that could change EN. The wait statement occurs at the end of the loop so that the EN signal is initially generated. This is how a concurrent signal assignment is done with SpecCharts, which only permits sequential code. Note that if there was no delay involved with the EN signal as there is now, then the EN signal would always be equal to $CNT/ = LIM$ and thus could be replaced by this expression. Note that we ***have*** replaced the ENIT and CNT_CLR signals used in the [Arms89, LiGa89] VHDL models, since they do not simplify the SpecChart description.
- **Update_output**– Ensures the controlled counter's output CNT_OUT always reflects the internal count CNT (essentially a concurrent signal assigment).
- **Counter**– This is the most interesting state as it performs the actual counting. Counter can be thought of as always being in one of two states, either performing count operations or performing an asynchronous clear. It thus consists of two sequential substates, Count and Clear. The default initial state is Count: if at any time CONSIG is "0001", CNT must be immediately cleared (actually after a small delay). This is accomplished by an Exit-immediately arc flowing to state Clear, which resets the count value to 0. The 'after 5 ns' clause in Clear not only delays the clearing for 5 ns, but also creates a 5 ns hold time for the clear to occur (otherwise by definition of an EI arc CNT would not get updated).

When the Count state is active, it can be in one of three states, either counting up, counting down, or waiting for the next count operation. Note that wait_state demonstrates one use of a state with a single null statement. This is a common occurence in SpecCharts.

From the above description, it is clear that the signals CONSIG, LIM, CNT, and EN must be declared somewhere. They are declared along with the ports in the topmost state, Controlled_counter, whose declarations are global over all substates. The declarations assume the existence of a *nibble* subtype, declared as bit_vector(3 downto 0). This is included in a package that is passed to the translator and thus added to the final VHDL code.

1.3 Alternative Model

A model was also developed which was identical to that given above except for the Counter state. The Counter state's functionality was described with code, rather than with sequential substates. The code was similar to that found in block CNT_UP_OR_DOWN in [Arms89, LiGa89] (see figure 3 for Counter's code). This resulted in less SpecChart code, but we feel using sequential substates and arcs enhances the understandability of the model.

1.4 Simulation and Results

The SpecChart description was simulated to verify correctness. Since the full graphical interface for SpecCharts does not exist, we entered the design using our current graphical interface, an X widget based system. The design is then written to a file in a completely textual representation. We then invoke our SpecChart to VHDL translator, which automatically generates a simulatable VHDL entity (included with this report). We create a new vhdl file which merely instantiates the counter entity and then drive its ports, checking for correct output. The stimulus file used was written to not just test our controlled counter specification, but also all other CADLAB VHDL descriptions of the controlled counter. The textual SpecChart code, stimulus file, simulation output, and the VHDL code generated by the translator are all included in the appendices of this report. Note that the simulation output shows that all of the self-checking assertions in the stimulus file were successful. Simulation results were identical for both models introduced above.

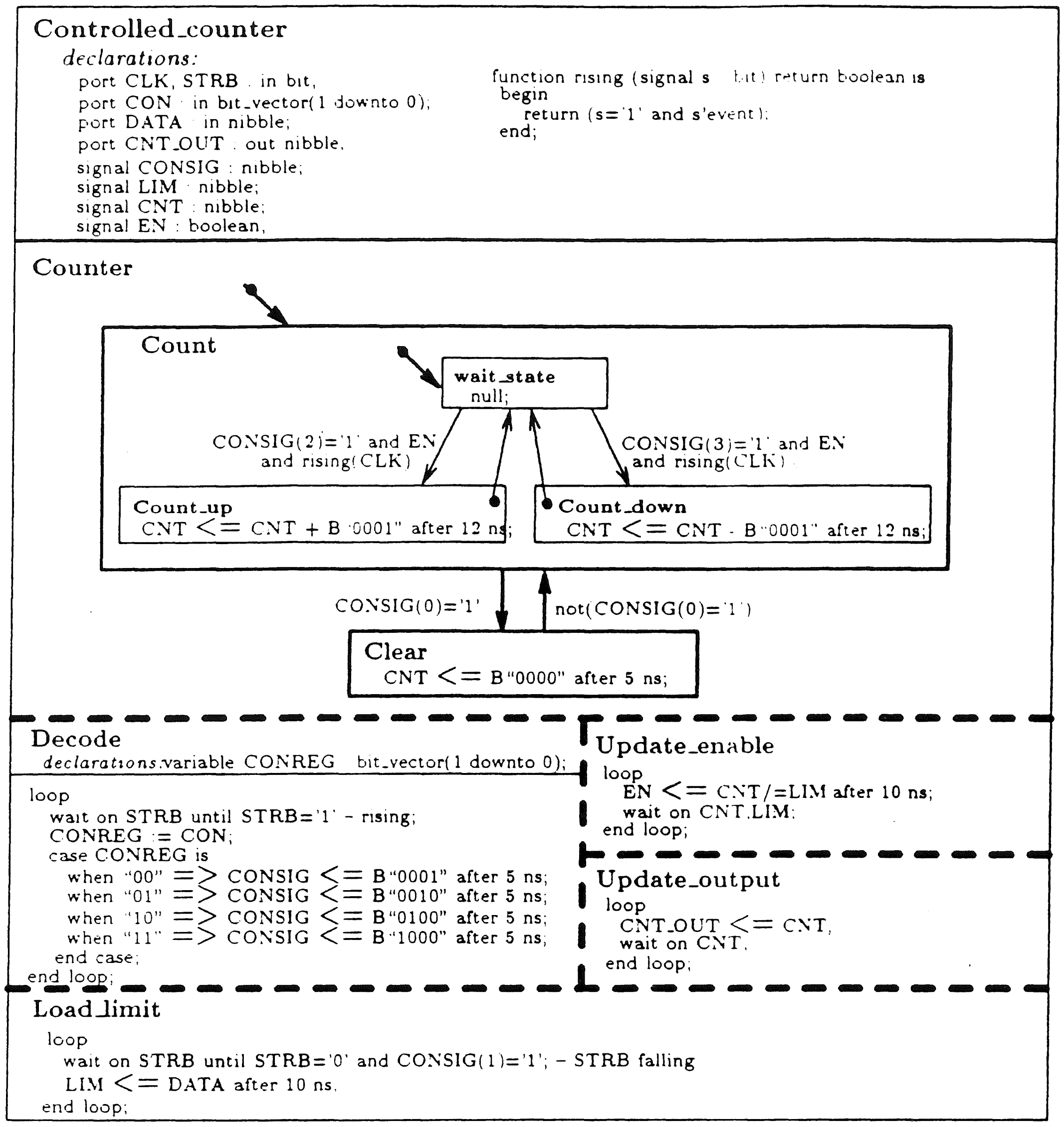
The SpecChart to VHDL translator ran in .3 seconds on a Sun 4. Table 1 shows the number of lines of text needed for various models, including the VHDL generated automatically by the translator. The bigger size of the generated VHDL code is attributable to a large extent to the sequential substate based SpecChart description, as opposed to the handwritten dataflow and process descriptions. This is verified by the shorter length of the alternative SpecChart's generated VHDL code.

SpecCharts are intended for system level specification and synthesis. The example discussed in this section is certainly not system level, and it is thus questionable whether or not one would want to intensively use SpecCharts at this level. However, two important points concerning SpecCharts are demonstrated by the example. First, synthesis will add much detail to the specification, so the language should have the ability to represent this detail.

Model	*Lines of code*
SpecChart model (sequential substates Counter)	67
Armstrong's mixed block/process description	81
Lis' dataflow description	52
Lis' process description	71
SpecChart textual files (created by SpecChart X application)	139
VHDL generated by SpecChart to VHDL translator	271
Alternative SpecChart model (using code for Counter)	57
Alternative SpecChart textual files	105
VHDL generated by translator for alternative model	158

Table 1: Lines of code for various Controlled Counter models, including generated VHDL

Second, a SpecChart description is easier to understand than a VHDL description, so might be useful when ease of understanding is important.



```
Controlled_counter
  declarations:
    port CLK, STRB : in bit;
    port CON : in bit_vector(1 downto 0);
    port DATA : in nibble;
    port CNT_OUT : out nibble;
    signal CONSIG : nibble;
    signal LIM : nibble;
    signal CNT : nibble;
    signal EN : boolean;

  function rising (signal s : bit) return boolean is
  begin
    return (s='1' and s'event);
  end;
```

```
wait_state
  null;

Count_up
  CNT <= CNT + B"0001" after 12 ns;

Count_down
  CNT <= CNT - B"0001" after 12 ns;

Clear
  CNT <= B"0000" after 5 ns;
```

```
Decode
  declarations:variable CONREG : bit_vector(1 downto 0);
loop
  wait on STRB until STRB='1' - rising;
  CONREG := CON;
  case CONREG is
    when "00" => CONSIG <= B"0001" after 5 ns;
    when "01" => CONSIG <= B"0010" after 5 ns;
    when "10" => CONSIG <= B"0100" after 5 ns;
    when "11" => CONSIG <= B"1000" after 5 ns;
  end case;
end loop;
```

```
Update_enable
  loop
    EN <= CNT/=LIM after 10 ns;
    wait on CNT,LIM;
  end loop;
```

```
Update_output
  loop
    CNT_OUT <= CNT;
    wait on CNT;
  end loop;
```

```
Load_limit
  loop
    wait on STRB until STRB='0' and CONSIG(1)='1'; - STRB falling
    LIM <= DATA after 10 ns;
  end loop;
```

Figure 2: SpecChart description of the Controlled Counter

Counter

```
loop
   wait until CONSIG(0)='1' or CLK='1';
   if CONSIG(0)='1' then
      CNT <= B"0000" after 5 ns;
   elsif CONSIG(2)='1' and EN then
      CNT <= CNT + B"0001" after 12 ns;
   elsif CONSIG(3)='1' and EN then
      CNT <= CNT - B"0001" after 12 ns;
   end if;
end loop;
```

Figure 3: Alternative SpecChart description of state Counter

2 Modeling DRACO using SpecCharts

This section describes the modeling of the 1781 discrete I/O backplane custom integrated circuit, DRACO, using the SpecCharts language. DRACO is described and the model is explained, followed by the simulation results.

2.1 DRACO Description and the SpecChart Model

The block diagram of the DRACO chip is shown in Figure 4. The SpecChart model of the DRACO chip is shown in Figure 5. Details of the operation of the DRACO chip can be found in [GuDu90, Rock89]. The SpecChart consists of an two-level hierarchy of states.

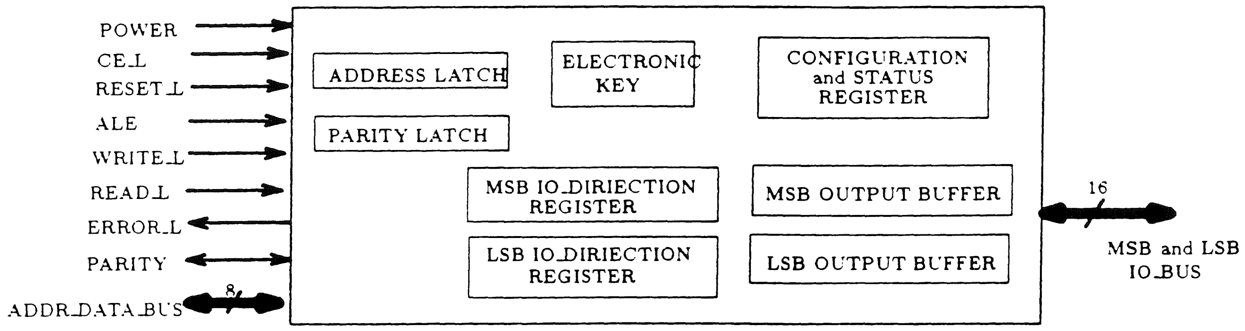


Figure 4: Block Diagram of the DRACO chip

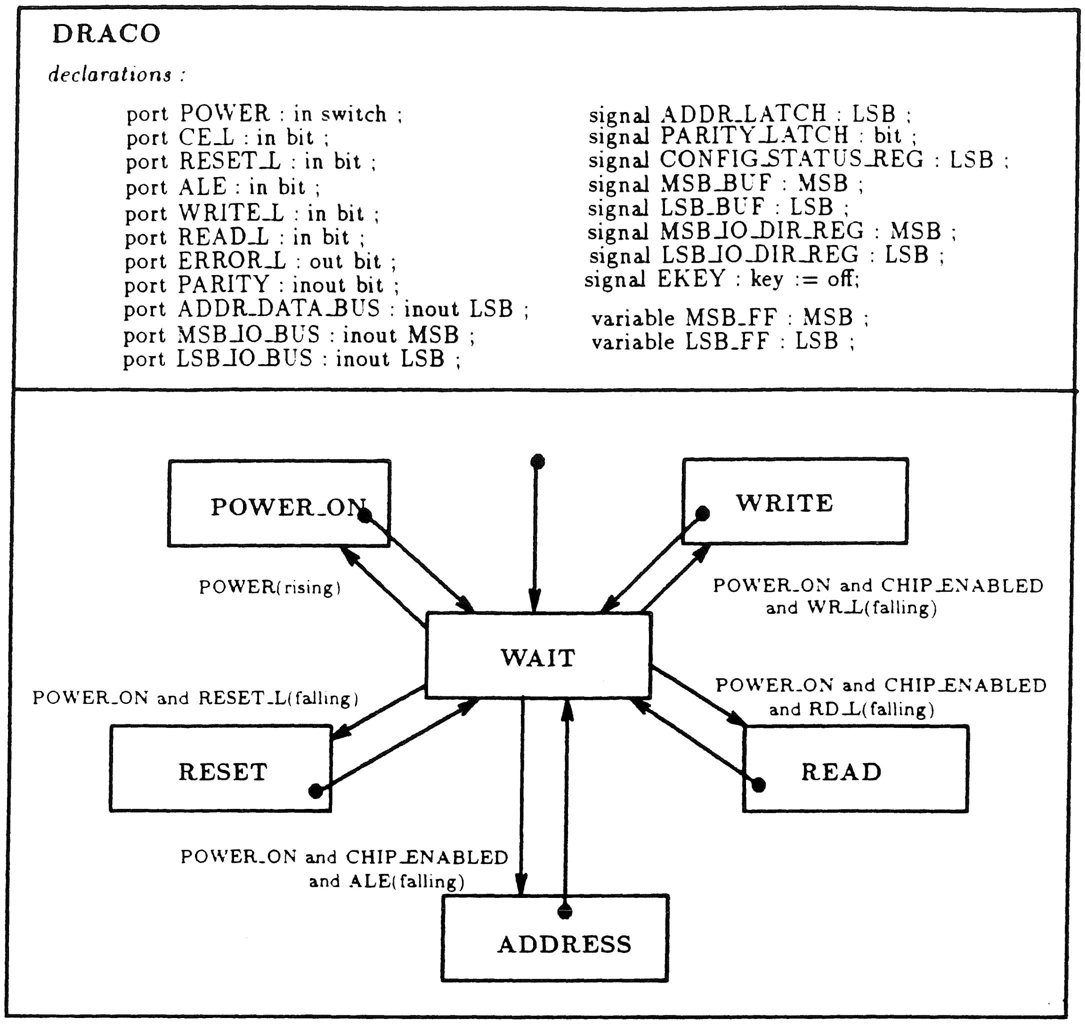
The state DRACO_TOP consists of six *sequential substates* - POWER_ON, RESET, ADDRESS, READ, WRITE, and a WAIT state. Each of these states is a *leaf* state i.e. have no further substates but contain sequential VHDL statements. The actions carried out in each of these states are discussed below.

The following types have been defined and made available to the DRACO SpecChart -

```
type switch    is (off,onn) ;
type key       is (off, mid, onn);

subtype MSB    is BIT_VECTOR (15 downto 8)  ;
subtype LSB    is BIT_VECTOR (7 downto 0)   ;
```

2.1.1 State DRACO_TOP



DRACO

declarations :

```
port POWER : in switch ;
port CE_L : in bit ;
port RESET_L : in bit ;
port ALE : in bit ;
port WRITE_L : in bit ;
port READ_L : in bit ;
port ERROR_L : out bit ;
port PARITY : inout bit ;
port ADDR_DATA_BUS : inout LSB ;
port MSB_IO_BUS : inout MSB ;
port LSB_IO_BUS : inout LSB ;

signal ADDR_LATCH : LSB ;
signal PARITY_LATCH : bit ;
signal CONFIG_STATUS_REG : LSB ;
signal MSB_BUF : MSB ;
signal LSB_BUF : LSB ;
signal MSB_IO_DIR_REG : MSB ;
signal LSB_IO_DIR_REG : LSB ;
signal EKEY : key := off;

variable MSB_FF : MSB ;
variable LSB_FF : LSB ;
```

POWER_ON
WRITE
POWER(rising)
POWER_ON and CHIP_ENABLED and WR_L(falling)
WAIT
POWER_ON and RESET_L(falling)
POWER_ON and CHIP_ENABLED and RD_L(falling)
RESET
READ
POWER_ON and CHIP_ENABLED and ALE(falling)
ADDRESS

Figure 5: SpecChart of the DRACO chip (only top-level shown)

The outermost state, DRACO_TOP, contains all the declarations of the external ports and internal data structure like registers and latches. The following ports are defined for communication externally -

POWER	Port indicating power-up of the chip. While POWER does not exist as a pin on the DRACO chip, to keep the model simple, it has been declared as a port to replace the VDD0, VDD1, VSS1 and the 6 VSS0 pins. The POWER port has type *switch.*
CE_L	Chip Enable pin
RESET_L	Reset pin

ALE	Address Latch Enable pin
WRITE_L	Write pin
RD_L	Read pin
ERROR_L	Error pin
PARITY	Parity pin (bidirectional)
ADDR_DATA_BUS	8 bit bidirectional bus
MSB_IO_BUS	High order byte of the 16 bit bidirectional IO bus
LSB_IO_BUS	Low order byte of the 16 bit bidirectional IO bus

The DRACO_TOP state also declares certain registers and latches, internal to the DRACO chip. These data structures are -

ADDR_LATCH	Address latch, stores the address within DRACO which will be written to or read from.
PARITY_LATCH	Parity Latch, stores the parity of the address stored
CONFIG_STATUS_REG	ConfigurationStatus register, used to write and read error-checking enabling operations and status information.
MSB_BUF, LSB_BUF	High and Low order byte output buffers, store data temporarily when checksum error-checking operations are enabled.
MSB_FF, LSB_FF	These flip flops hold the inverted value of the output buffers during a write operation
MSB_IO_DIR_REG, LSB_IO_DIR_REG	High and Low order byte direction registers, used to set DRACO ports for read-only or bidirectional operation
EKEY	3-stage Electronic Key, prevents unauthorized applications of DRACO and ensures integrity of configuration and direction registers.

2.1.2 State WAIT

This state is the initial substate of the state DRACO_TOP. As the name implies it only serves as a temporary state for transitions between the other substates (POWER, RESET, ADDRESS, WRITE and READ). On completion, each of these states transfer control to the state WAIT.

2.1.3 State POWER_ON

This state waits for power-up and then sets the ADDR_LATCH to FFH, and clears the IO direction registers.

2.1.4 State RESET

The RESET state waits for a low on the RESET_L pin of the DRACO. It then clears the latches ADDR_LATCH and PARITY_LATCH, the status register CONFIG_STATUS_REG, the output buffers MSB_BUF and LSB_BUF, and the direction registers MSB_IO_DIR_REG and LSB_IO_DIR_REG.

The ERROR_L pin is set to high. The electronic key, EKEY, is set to the "off" position.

2.1.5 State ADDRESS

The ADDRESS state latches the ADDR_LATCH and the PARITY_LATCH on a high to low transition on the ALE pin of the DRACO chip.

2.1.6 State WRITE

The WRITE state waits for a low ro high transition on the WR_L pin. It then examines the latched address for a parity error if the address parity checking option had previously been enabled. Next, if the data parity checking option of the DRACO was enabled, the parity of the data byte received is checked.

The WRITE state does the following, depending upon the address being written to -

80H	If the data byte is AAH and EKEY is not already "on", it is set to position "mid". If however the data byte is not AAH, the EKEY is set to "off".
7FH	If EKEY is "on", if the data byte is 55H, then the data is unlocked else if the data byte being written is AAH, then the configuration is unlocked. If the EKEY was in the "mid" setting and the data byte is 55H, then the EKEY is set to "on" else it is set to the "off" position.
0FH	Clears the latched interrupt, the ERROR_L pin is set to '1'.
04H	If the EKEY is in the unlocked configuration position, the high order byte of the IO direction register is written to. This configures the high order IO ports as bidirectional or input only. The data on the ports
03H	If the EKEY is in the unlocked configuration position, the low order byte of the IO direction register is written to. This configures the low order IO ports as bidirectional or input only.
02H	If ther EKEY is in the unlock configuration position, a write to this address will cause the three lower bits of the data byte to be written into the configuration register . These bits select the error checking options.
01H	If the checksum error checking is enabled, the data byte will be written to the high order byte output buffer, MSB_BUF, else to the high order byte IO ports will be updated. Since the IO ports are active low, the value of the data in the output buffers is inverted
00H	If the checksum error checking is enabled, the data byte will be written to the low order byte output buffer, LSB_BUF, else to the low order byte IO ports will be updated. Since the IO ports are active low, the value of the data in the output buffers is inverted

0EH	If the checksum error checking is unabled, the inverted checksum of the date in the output buffers is computed and compared with the checksum byte being written. If they are equal, the IO ports are updated from the buffers.

The WRITE state will generate error in the following cases -

- - Parity error on a write address
- - Parity error on write data
- - Write to an invalid address
- - Write to an address locked by the EKEY
- - Invalid checksum write, with checksum mode enabled
- - Write to checksum address, with checksum mode disabled

In case of an error, the ERROR_L pin is cleared. In all the write operations, the appropriate bits of the CONFIG_STATUS_REG are constantly updated. For further details, refer to the DRACO data sheet. Even in the where the checksum error checking option is disabled, a write to the high or low order ports will also cause the output buffers (MSB_BUF, LSB_BUF) and the flip flops (MSB_FF, LSB_FF) to be updated too.

2.1.7 State READ

A high to low transition on the RD_L pin activates the READ state. The READ state then examines the latched address for a parity error if the addre ss parity checking option had previously been enabled. Next, depending on the address of the read operation, the READ state does the following -

0EH	Read the inverted checksum of the high and low bytes of the output buffers
04H	Read the high order byte of the output buffer
03H	Read the low order byte of the output buffer
02H	Read the status register, CONFIG_STATUS_REG
01H	Read the high byte of the IO port
00H	Read the low byte of the IO port

All the data being read is routed through an internal data bus (INTERNAL_DBUS) before being output to the ADDR_DATA_BUS. In all the above read operations, the READ state also places the appropriate parity bit on the PARITY pin. If an attempt is made from an invalid address, DRACO will output its internal data bus with an incorrect parity on the PARITY pin. The READ state will generate error in the following cases -

- - Parity error on a read address
- - Read from an invalid address.

The complete SpecChart model of the DRACO is given in appendix B.

2.2 Simulation and Results

The SpecChart was entered in the same manner as the Controlled counter. The test pattern file (approx. 23,000 lines) which was used to verify the completeness and correctness of the SpecChart model of DRACO was supplied by Rockwell Corporation. The test vectors were identical to the ones used by Rockwell to test the chip designed by them. Since the test vectors were in the VTI format, a parser was developed to translate it to VHDL process sequential statements.

A VHDL file instantiated the DRACO entity produced by the translator and used the test vectors to drive its ports, checking the outputs for correct behavior. The results of the simulation confirmed the completeness and the accuracy of the SpecChart model of DRACO.

Model	*Lines of code*
SpecChart model of DRACO	226
[GuDu90] VHDL description of DRACO	392
SpecChart textual files of DRACO	268
VHDL entity generated by SpecChart to VHDL translator for DRACO	506

Table 2: Lines of code for various DRACO models, including generated VHDL

The SpecChart code for the DRACO is in appendix B. Table 2 shows the number of lines needed for various DRACO models, including the automatically generated VHDL model. The number of lines of SpecChart code needed to specify the data structures needed, functionality of the states, and state transitions was 226. A manually generated VHDL description [GuDu90] of the DRACO had 392 lines of VHDL code. Thus unlike the handwritten VHDL model, the SpecChart specification was more concise since control and sequencing information did not need to be explicitly specified.

Howevever, when the DRACO SpecChart was translated to VHDL, the resulting code consisted of 506 lines. Thus while a SpecChart description may be more concise than a pure VHDL description, the code produced by the SC_toVHDL translator may be large. The translation of the SpecChart to VHDL took approximately 3 seconds. The compilation on Zycad simulator of the DRACO entity took 3 seconds, and compilation of the 23,000 line Rockwell test vector file required 9 minutes. The simulation then takes about 2 minutes.

3 Conclusion

This report provided two detailed examples of SpecChart models. The examples proved that SpecCharts can be used to concisely model designs, which can then be verified via the VHDL translator. They demonstrated that the concept of a high-level language built on top of VHDL is beneficial for the modeler and does not decrease the efficiency of the simulation. Neither of the models were at the system level, and thus the differences in the sizes of the SpecChart models compared to the handwritten VHDL models were not extremely large. However, even at the given level, the SpecChart models were somewhat more concise, and we feel much easier to understand.

4 Acknowledgements

This work was supported by the National Science Foundation (grant #MIP-8922851) and the Semiconductor Research Corporation (grant #89-DJ-146). We are grateful for their support. The authors of ths report would like to thank Bob Larsen, Johnny Sitou, and Dave Pasella of Rockwell Corporation for their help in providing us the details of the DRACO chip and making available the DRACO test vector file. We would also like to thank Joe Lis, Tedd Hadley, and Rajesh Gupta for their advice and suggestions.

5 References

[Arms89] Armstrong, J., "Chip Level Modeling Using VHDL", Prentice-Hall, 1989.

[GuDu90] Gupta, R., and Dutt, N., "Behavioral Modeling of DRACO: A Peripheral Interface ASIC", University of California, Irvine, Technical Report 90-13, June 1990.

[LiGa89] Lis, J., and Gajski, D.D., "Structured Modeling for VHDL Synthesis", University of California, Irvine, Technical Report 89-14, June 1989.

[Rock89] Rockwell International, "DRACO Engineering Report", April 1989.

[VaNaGa90a] Vahid, F., Narayan, S., and Gajski, D.D., "Synthesis from Specifications", University of California, Irvine, Technical Report 90-03, January 1990.

[VaNaGa90b] Vahid, F., Narayan, S., and Gajski, D.D., "SpecCharts: A Language for System Level Specification and Synthesis", University of California, Irvine, Technical report 90-19, July 1990.

A Controlled Counter Appendix

A.1 Controlled Counter SpecChart textual code

```
state
{
  name {Clear}
  code
  {  CNT <= B"0000" after 5 ns; }
}

state
{
  name {Controlled_counter}
  declarations
  {
     port CLK : in bit;
     port STRB : in bit;
     port CON : in bit_vector(1 downto 0);
     port DATA : in nibble;
     port CNT_OUT : out nibble;
     signal CONSIG : nibble;
     signal LIM : nibble;
     signal CNT : nibble;
     signal EN  : boolean;
     function rising_fct (signal s : bit) return boolean is
      begin
        return (s='1' and s'event);
      end;
  }
  concurrent substates
  {
    Counter : ;
    Decode : ;
    Load_limit : ;
    Update_output : ;
    Update_enable : ;
  }
}

state
{
  name {Count}
  sequential substates
  {
     wait_state :
        (EI, CONSIG(2)='1' and EN and rising_fct(CLK), Count_up),
        (EI, CONSIG(3)='1' and EN and rising_fct(CLK), Count_down);
     Count_up :     (EOC, true, wait_state);
     Count_down :   (EOC, true, wait_state);
  }
}

state
{
  name {Count_down}
  code
  {  CNT <= CNT - B"0001" after 12 ns;
}

}

state
{
  name {Count_up}
  code
```

```
    {  CNT <= CNT + B"0001" after 12 ns;}
  }

  state
  {
    name {Counter}
    sequential substates
    {  Count : (EI, CONSIG(0)='1', Clear);
       Clear : (EI, not(CONSIG(0)='1'), Count);
    }
  }

  state
  {
    name {Decode}
    declarations
    {  variable CONREG : bit_vector(1 downto 0);}
    code
    { loop
        wait on STRB until STRB='1';
        CONREG := CON;
        case CONREG is
           when "00" =>
             CONSIG <= B"0001" after 5 ns;
           when "01" =>
     CONSIG <= B"0010" after 5 ns;
           when "10" =>
             CONSIG <= B"0100" after 5 ns;
           when "11" =>
             CONSIG <= B"1000" after 5 ns;
        end case;
      end loop;
    }
  }

  state
  {
    name {Load_conval}
    code
    { loop
        wait on STRB, CON until STRB='1';
        CONVAL <= CON after 5 ns;
      end loop;
    }
  }

  state
  {
    name {Load_limit}
    code
    {  loop
         wait on STRB until STRB='0' and CONSIG(1)='1';
         LIM <= DATA after 10 ns;
       end loop;
    }
  }

  state
  {
    name {Update_enable}
    code
    {loop
       EN <= CNT/=LIM after 10 ns;
       wait on CNT,LIM;
    end loop;}
  }

  state
```

```
{
  name {Update_output}
  code
  { loop
      CNT_OUT <= CNT;
      wait on CNT;
     end loop;
  }
}

state
{
  name {wait_state}
  code
  {null; }
}
```

A.2 Controlled Counter Test Stimulus File

```
-------------------------------------------------------------------------------
-- file: sim.vhd
-- authors: Frank Vahid, Sanjiv Narayan
-- desc: provides simple functionality verification of the Controlled counter
--       uses bit_vector inputs, requires bit functions package
-- notes: * This file was originally written for models using integers instead
--          of bit vectors.  It has been converted to work for bit vectors, but
--          therefore looks a little funny since it tracks the integers and
--          converts them, and vice-versa.
--        * uses 'downto' bit vector direction, the agreed upon CADLAB standard.

-- date: 6/20/90
-------------------------------------------------------------------------------


use work.bit_functions.all;

entity E is
end;

architecture A of E is
   component Controlled_counterE
      port (
            CLK      : in bit;
            STRB     : in bit;
            CON      : in bit_vector(0 to 1);
            DATA     : in bit_vector(0 to 3);
            CNT_OUT  : out bit_vector(0 to 3)
   );
   end component;

   signal CLK       : bit;
   signal STRB      : bit;
   signal CON       : integer range 0 to 3;
   signal DATA      : integer range 0 to 15;
   signal CNT_OUT   : integer range 0 to 15;
   signal CONbv       : bit_vector(0 to 1);
   signal DATAbv      : bit_vector(0 to 3);
   signal CNT_OUTbv   : bit_vector(0 to 3);



   for all : Controlled_counterE
       use entity work.Controlled_counterE(Controlled_counterA);

begin
   CC : Controlled_counterE port map (CLK, STRB, CONbv, DATAbv, CNT_OUTbv);

   -- track the integers/bit_vectors, convert to bit_vectors/integers
   CONbv <= INT_TO_BIN(CON,2);
   DATAbv <= INT_TO_BIN(DATA,4);
   CNT_OUT <= BIN_TO_INT(CNT_OUTbv);

   process
   begin
      wait for 1 ns;
      CLK <= transport '0';
      wait for 49 ns;
      CLK <= transport '1';
   end process;

   process
   begin

      wait for 30 ns;
```

```
-- start off with simple test of reset, count up, and count down, and limit

     -- reset the counter
     CON <= 0;
     STRB <= '1' after 10 ns, '0' after 20 ns;
     wait for 50 ns;
     assert (CNT_OUT=0) report "ERROR1: CNT_OUT not reset to 0";
     -- t=80

     -- load the LIMIT
     DATA <= 2;
     CON <= 1;
     STRB <= '1' after 10 ns, '0' after 20 ns;
     wait for 50 ns;
     -- t=130

     -- count up
     CON <= 2;
     STRB <= '1' after 10 ns, '0' after 20 ns;
     wait for 50 ns;
     assert (CNT_OUT=1) report "ERROR2: CNT_OUT not incremented to 1";
     -- t=180

     -- count up again
     wait for 50 ns;
     assert (CNT_OUT=2) report "ERROR3: CNT_OUT not incremented to 2";
     -- t=230

     -- count up, should not increment since hit limit
     wait for 50 ns;
     assert (CNT_OUT=2) report "ERROR4: CNT_OUT should have hit limit at 2";
     -- t=280

     -- count down, should not decrement since hit limit
     CON <= 3;
     STRB <= '1' after 10 ns, '0' after 20 ns;
     wait for 50 ns;
     assert (CNT_OUT=2) report "ERROR5: CNT_OUT at limit, shouldn't change";
     -- t=330

     -- load the LIMIT
     DATA <= 0;
     CON <= 1;
     STRB <= '1' after 10 ns, '0' after 20 ns;
     wait for 50 ns;
     -- t=380

     -- count down
     CON <= 3;
     STRB <= '1' after 10 ns, '0' after 20 ns;
     wait for 50 ns;
     assert (CNT_OUT=1) report "ERROR6: CNT_OUT not decremented to 1";
     -- t=430

-- do some extensive testing of the counter's limit handling
     -- set limit to 13
     DATA <= 13;
     CON <= 1;
     STRB <= '1' after 10 ns, '0' after 20 ns;
     wait for 50 ns;
     -- t=480

     -- reset the counter
     CON <= 0;
     STRB <= '1' after 10 ns, '0' after 20 ns;
     wait for 50 ns;
     assert (CNT_OUT=0) report "ERROR7: CNT_OUT not reset to 0";
```

```
-- t=530

-- count up to 13
CON <= 2;
STRB <= '1' after 10 ns, '0' after 20 ns;
for i in 1 to 13 loop
   wait for 50 ns;
end loop;
-- t=1180
assert (CNT_OUT=13) report "ERROR8: CNT_OUT not up to 13";

-- count up, should not increment since hit limit
wait for 50 ns;
assert (CNT_OUT=13) report "ERROR9: CNT_OUT should have hit limit at 13";
-- t=1230

-- count up, should not increment since hit limit
wait for 50 ns;
assert (CNT_OUT=13) report "ERROR10: CNT_OUT should have hit limit at 13";

-- t=1280

-- change limit to 15
DATA <= 15;
CON <= 1;
STRB <= '1' after 10 ns, '0' after 20 ns;
wait for 50 ns;
-- t=1330

-- count up
CON <= 2;
STRB <= '1' after 10 ns, '0' after 20 ns;
wait for 50 ns;
assert (CNT_OUT=14) report "ERROR11: CNT_OUT didn't increment to 14";
-- t=1380

-- count up
wait for 50 ns;
assert (CNT_OUT=15) report "ERROR12: CNT_OUT didn't increment to 15";
-- t=1430

-- count up, should not increment since hit limit
wait for 50 ns;
assert (CNT_OUT=15) report "ERROR13:CNT_OUT should have hit limit at 15";
-- t=1480

-- change limit to 7
DATA <= 7;
CON <= 1;
STRB <= '1' after 10 ns, '0' after 20 ns;
wait for 50 ns;
-- t=1530

-- count down, try counting below 7
CON <= 3;
STRB <= '1' after 10 ns, '0' after 20 ns;
for i in 1 to 10 loop
   wait for 50 ns;
end loop;
assert (CNT_OUT=7) report "ERROR14: CNT_OUT not equal to 7";
-- t=2030

-- change limit to 0
DATA <= 0;
CON <= 1;
STRB <= '1' after 10 ns, '0' after 20 ns;
wait for 50 ns;
-- t=2080
```

```
    -- count down, try counting below 8
    CON <= 3;
    STRB <= '1' after 10 ns, '0' after 20 ns;
    for i in 1 to 8 loop
       wait for 50 ns;
    end loop;
    assert (CNT_OUT=0) report "ERROR15: CNT_OUT not equal to 0";
    -- t=2480

    -- count up, should not increment since hit limit
    CON <= 2;
    STRB <= '1' after 10 ns, '0' after 20 ns;
    wait for 50 ns;
    assert (CNT_OUT=0) report "ERROR16: CNT_OUT should have stayed at 0";
    -- t=2530

-- try counting beyond the range, i.e. above 15 and below 0

    -- reset the counter
    CON <= 0;
    STRB <= '1' after 10 ns, '0' after 20 ns;
    wait for 50 ns;
    assert (CNT_OUT=0) report "ERROR17: CNT_OUT not reset to 0";
    -- t=2580

    -- change limit to 7
    DATA <= 7;
    CON <= 1;
    STRB <= '1' after 10 ns, '0' after 20 ns;
    wait for 50 ns;
    -- t=2630

    -- count up 1
    CON <= 2;
    STRB <= '1' after 10 ns, '0' after 20 ns;
    wait for 50 ns;
    assert (CNT_OUT=1) report "ERROR18: CNT_OUT not incremented to 1";
    -- t=2680

    -- count down
    CON <= 3;
    STRB <= '1' after 10 ns, '0' after 20 ns;
    wait for 50 ns;
    assert (CNT_OUT=0) report "ERROR19:CNT_OUT not decremented to 0";
    -- t=2730

    -- count down
    wait for 50 ns;
    assert (CNT_OUT=15) report "ERROR20: CNT_OUT not decremented to 15";
    -- t=2780

    -- count down
    wait for 50 ns;
    assert (CNT_OUT=14) report "ERROR21: CNT_OUT not decremented to 14";
    -- t=2830

    -- count up
    CON <= 2;
    STRB <= '1' after 10 ns, '0' after 20 ns;
    wait for 50 ns;
    assert (CNT_OUT=15) report "ERROR22: CNT_OUT not incremented to 15";
    -- t=2880

    -- count up
    CON <= 2;
    STRB <= '1' after 10 ns, '0' after 20 ns;
    wait for 50 ns;
```

```
      assert (CNT_OUT=0) report "ERROR23: CNT_OUT not incremented to 0";
      -- t=2930

   wait;
   end process;
end A;
```

A.3 Controlled Counter Simulation Output

```
0 NS
   SMON3:     ACTIVE /E/CNT_OUT (value = 0)
40 NS
    SMON:     ACTIVE /E/STRB (value = '1')
   SMON1:     ACTIVE /E/CON (value = 0)
80 NS
    SMON:     ACTIVE /E/STRB (value = '1')
   SMON1:     ACTIVE /E/CON (value = 1)
   SMON2:     ACTIVE /E/DATA (value = 2)
90 NS
    SMON:     ACTIVE /E/STRB (value = '0')
130 NS
    SMON:     ACTIVE /E/STRB (value = '1')
   SMON1:     ACTIVE /E/CON (value = 2)
162 NS
   SMON3:     ACTIVE /E/CNT_OUT (value = 1)
212 NS
   SMON3:     ACTIVE /E/CNT_OUT (value = 2)
280 NS
    SMON:     ACTIVE /E/STRB (value = '1')
   SMON1:     ACTIVE /E/CON (value = 3)
330 NS
    SMON:     ACTIVE /E/STRB (value = '1')
   SMON1:     ACTIVE /E/CON (value = 1)
   SMON2:     ACTIVE /E/DATA (value = 0)
340 NS
    SMON:     ACTIVE /E/STRB (value = '0')
380 NS
    SMON:     ACTIVE /E/STRB (value = '1')
   SMON1:     ACTIVE /E/CON (value = 3)
412 NS
   SMON3:     ACTIVE /E/CNT_OUT (value = 1)
430 NS
    SMON:     ACTIVE /E/STRB (value = '1')
   SMON1:     ACTIVE /E/CON (value = 1)
   SMON2:     ACTIVE /E/DATA (value = 13)
440 NS
    SMON:     ACTIVE /E/STRB (value = '0')
490 NS
    SMON:     ACTIVE /E/STRB (value = '1')
   SMON1:     ACTIVE /E/CON (value = 0)
500 NS
   SMON3:     ACTIVE /E/CNT_OUT (value = 0)
530 NS
    SMON:     ACTIVE /E/STRB (value = '1')
   SMON1:     ACTIVE /E/CON (value = 2)
562 NS
   SMON3:     ACTIVE /E/CNT_OUT (value = 1)
612 NS
   SMON3:     ACTIVE /E/CNT_OUT (value = 2)
662 NS
   SMON3:     ACTIVE /E/CNT_OUT (value = 3)
712 NS
   SMON3:     ACTIVE /E/CNT_OUT (value = 4)
762 NS
   SMON3:     ACTIVE /E/CNT_OUT (value = 5)
812 NS
   SMON3:     ACTIVE /E/CNT_OUT (value = 6)
862 NS
   SMON3:     ACTIVE /E/CNT_OUT (value = 7)
912 NS
   SMON3:     ACTIVE /E/CNT_OUT (value = 8)
962 NS
   SMON3:     ACTIVE /E/CNT_OUT (value = 9)
1012 NS
```

```
   SMON3:     ACTIVE /E/CNT_OUT (value = 10)
1062 NS
   SMON3:     ACTIVE /E/CNT_OUT (value = 11)
1112 NS
   SMON3:     ACTIVE /E/CNT_OUT (value = 12)
1162 NS
   SMON3:     ACTIVE /E/CNT_OUT (value = 13)
1280 NS
    SMON:     ACTIVE /E/STRB (value = '1')
   SMON1:     ACTIVE /E/CON (value = 1)
   SMON2:     ACTIVE /E/DATA (value = 15)
1290 NS
    SMON:     ACTIVE /E/STRB (value = '0')
1330 NS
    SMON:     ACTIVE /E/STRB (value = '1')
   SMON1:     ACTIVE /E/CON (value = 2)
1362 NS
   SMON3:     ACTIVE /E/CNT_OUT (value = 14)
1380 NS
    SMON:     ACTIVE /E/STRB (value = '0')
1412 NS
   SMON3:     ACTIVE /E/CNT_OUT (value = 15)
1480 NS
    SMON:     ACTIVE /E/STRB (value = '1')
   SMON1:     ACTIVE /E/CON (value = 1)
   SMON2:     ACTIVE /E/DATA (value = 7)
1490 NS
    SMON:     ACTIVE /E/STRB (value = '0')
1530 NS
    SMON:     ACTIVE /E/STRB (value = '1')
   SMON1:     ACTIVE /E/CON (value = 3)
1562 NS
   SMON3:     ACTIVE /E/CNT_OUT (value = 14)
1612 NS
   SMON3:     ACTIVE /E/CNT_OUT (value = 13)
1662 NS
   SMON3:     ACTIVE /E/CNT_OUT (value = 12)
1712 NS
   SMON3:     ACTIVE /E/CNT_OUT (value = 11)
1762 NS
   SMON3:     ACTIVE /E/CNT_OUT (value = 10)
1812 NS
   SMON3:     ACTIVE /E/CNT_OUT (value = 9)
1862 NS
   SMON3:     ACTIVE /E/CNT_OUT (value = 8)
1912 NS
   SMON3:     ACTIVE /E/CNT_OUT (value = 7)
2030 NS
    SMON:     ACTIVE /E/STRB (value = '1')
   SMON1:     ACTIVE /E/CON (value = 1)
   SMON2:     ACTIVE /E/DATA (value = 0)
2040 NS
    SMON:     ACTIVE /E/STRB (value = '0')
2080 NS
    SMON:     ACTIVE /E/STRB (value = '1')
   SMON1:     ACTIVE /E/CON (value = 3)
2112 NS
   SMON3:     ACTIVE /E/CNT_OUT (value = 6)
2162 NS
   SMON3:     ACTIVE /E/CNT_OUT (value = 5)
2212 NS
   SMON3:     ACTIVE /E/CNT_OUT (value = 4)
2262 NS
   SMON3:     ACTIVE /E/CNT_OUT (value = 3)
2312 NS
   SMON3:     ACTIVE /E/CNT_OUT (value = 2)
2362 NS
   SMON3:     ACTIVE /E/CNT_OUT (value = 1)
```

```
2412 ■S
   SMO■3:     ACTIVE /E/C■T_OUT (value = 0)
2480 ■S
    SMO■:     ACTIVE /E/STRB (value = '1')
   SMO■1:     ACTIVE /E/CO■ (value = 2)
2540 ■S
    SMO■:     ACTIVE /E/STRB (value = '1')
   SMO■1:     ACTIVE /E/CO■ (value = 0)
2580 ■S
    SMO■:     ACTIVE /E/STRB (value = '1')
   SMO■1:     ACTIVE /E/CO■ (value = 1)
   SMO■2:     ACTIVE /E/DATA (value = 7)
2590 ■S
    SMO■:     ACTIVE /E/STRB (value = '0')
2630 ■S
    SMO■:     ACTIVE /E/STRB (value = '1')
   SMO■1:     ACTIVE /E/CO■ (value = 2)
2662 ■S
   SMO■3:     ACTIVE /E/C■T_OUT (value = 1)
2680 ■S
    SMO■:     ACTIVE /E/STRB (value = '1')
   SMO■1:     ACTIVE /E/CO■ (value = 3)
2712 ■S
   SMO■3:     ACTIVE /E/C■T_OUT (value = 0)
2730 ■S
    SMO■:     ACTIVE /E/STRB (value = '1')
   SMO■1:     ACTIVE /E/CO■ (value = 3)
2762 ■S
   SMO■3:     ACTIVE /E/C■T_OUT (value = 15)
2780 ■S
    SMO■:     ACTIVE /E/STRB (value = '1')
   SMO■1:     ACTIVE /E/CO■ (value = 3)
2812 ■S
   SMO■3:     ACTIVE /E/C■T_OUT (value = 14)
2830 ■S
    SMO■:     ACTIVE /E/STRB (value = '1')
   SMO■1:     ACTIVE /E/CO■ (value = 2)
2862 ■S
   SMO■3:     ACTIVE /E/C■T_OUT (value = 15)
2880 ■S
    SMO■:     ACTIVE /E/STRB (value = '1')
   SMO■1:     ACTIVE /E/CO■ (value = 2)
2912 ■S
   SMO■3:     ACTIVE /E/C■T_OUT (value = 0)
2962 ■S
   SMO■3:     ACTIVE /E/C■T_OUT (value = 1)
3012 ■S
   SMO■3:     ACTIVE /E/C■T_OUT (value = 2)
3062 ■S
   SMO■3:     ACTIVE /E/C■T_OUT (value = 3)
3112 ■S
   SMO■3:     ACTIVE /E/C■T_OUT (value = 4)
3162 ■S
   SMO■3:     ACTIVE /E/C■T_OUT (value = 5)
3212 ■S
   SMO■3:     ACTIVE /E/C■T_OUT (value = 6)
3262 ■S
   SMO■3:     ACTIVE /E/C■T_OUT (value = 7)
10000 ■S
```

A.4 Controlled Counter VHDL Code Generated by SpecChart to VHDL Translator

```
entity Controlled_counterE is
   port(CLK : in bit ; STRB : in bit ; CON : in bit_vector (1 downto 0) ; DATA :
  in nibble ; CNT_OUT : out nibble );
end;


Architecture Controlled_counterA of Controlled_counterE is
   signal inControlled_counter   : boolean := false;
   -- NOTE: Controlled_counter's declarations (except variables) have been pulle
d up to here.
   type Controlled_counter_nibble_RES is array (natural range <>) of nibble;
   function Controlled_counter_nibble_RES_fct ( INPUT :  Controlled_counter_nibb
le_RES ) return nibble is
   begin
      assert (INPUT'length = 1) report "overdriven signal, type: Controlled_coun
ter_nibble_RES" severity warning;
      return INPUT(0);
   end;
   signal CONSIG : Controlled_counter_nibble_RES_fct nibble register;
   signal LIM : Controlled_counter_nibble_RES_fct nibble register;
   signal CNT : Controlled_counter_nibble_RES_fct nibble register;
   type Controlled_counter_boolean_RES is array (natural range <>) of boolean;
   function Controlled_counter_boolean_RES_fct ( INPUT :  Controlled_counter_boo
lean_RES ) return boolean is
   begin
      assert (INPUT'length = 1) report "overdriven signal, type: Controlled_coun
ter_boolean_RES" severity warning;
      return INPUT(0);
   end;
   signal EN : Controlled_counter_boolean_RES_fct boolean register;
   function rising_fct (signal s :  bit ) return boolean is
   begin
      return (s = '1' and s'event);
   end;
   signal CNT_OUT_sig : Controlled_counter_nibble_RES_fct nibble register;
   signal inCounter : boolean := false;
   signal inDecode : boolean := false;
   signal inLoad_limit : boolean := false;
   signal inUpdate_output : boolean := false;
   signal inUpdate_enable : boolean := false;
   function MAX_fct ( a :  time ;  b :  time ) return time is
   begin
      if (a > b) then
      return a;
      else
      return b;
      end if;
   end;
begin
  Controlled_counter: block
  begin
      Counter: block
          signal inCount : boolean := false;
          signal inClear : boolean := false;
      begin
          Count: block
              signal inwait_state : boolean := false;
              signal inCount_up : boolean := false;
              signal doneCount_up : boolean := false;
              signal inCount_down : boolean := false;
              signal doneCount_down : boolean := false;
          begin
              wait_state: block (inwait_state and not(inwait_state'stable))
```

```
begin
    code: process
    begin
    if guard then
    wait_state_Loop : loop
    null;
    exit wait_state_Loop;
    end loop wait_state_Loop;
    end if;
    wait  on guard;
    end process code;
end block wait_state;
Count_up: block (inCount_up and not(inCount_up'stable))
begin
    code: process
        variable REMAIN_TIME: time;
        variable GLOBAL_TIME: time;
    begin
    if guard then
    Count_up_Loop : loop
    REMAIN_TIME := 0 fs;
    CNT <=  CNT;
    REMAIN_TIME := MAX_fct(REMAIN_TIME,12 ns);
    CNT <=  CNT + B"0001" after 12 ns;
    wait  until not (inCount_up)  for REMAIN_TIME;
    if (not inCount_up ) then
    exit Count_up_Loop;
    end if;
    doneCount_up <= transport true;
    wait  until not (inCount_up) ;
    doneCount_up <= transport false;
    exit Count_up_Loop;
    end loop Count_up_Loop;
    end if;
    CNT <= transport null;
    wait  on guard;
    end process code;
end block Count_up;
Count_down: block (inCount_down and not(inCount_down'stable))
begin
    code: process
        variable REMAIN_TIME: time;
        variable GLOBAL_TIME: time;
    begin
    if guard then
    Count_down_Loop : loop
    REMAIN_TIME := 0 fs;
    CNT <=  CNT;
    REMAIN_TIME := MAX_fct(REMAIN_TIME,12 ns);
    CNT <=  CNT - B"0001" after 12 ns;
    wait  until not (inCount_down)  for REMAIN_TIME;
    if (not inCount_down ) then
    exit Count_down_Loop;
    end if;
    doneCount_down <= transport true;
    wait  until not (inCount_down) ;
    doneCount_down <= transport false;
    exit Count_down_Loop;
    end loop Count_down_Loop;
    end if;
    CNT <= transport null;
    wait  on guard;
    end process code;
end block Count_down;
control: process begin
      if (inCount and not(inCount'stable)) then
         inwait_state <= transport true;
      elsif (inCount=false and not(inCount'stable)) then
```

```
                        inwait_state <= transport false;
                        inCount_up <= transport false;
                        inCount_down <= transport false;
                     elsif (inwait_state and CONSIG(3) = '1' and EN and rising_f
ct(CLK)) then
                        inwait_state <= transport false;
                        inCount_down <= transport true;
                     elsif (inwait_state and CONSIG(2) = '1' and EN and rising_f
ct(CLK)) then
                        inwait_state <= transport false;
                        inCount_up <= transport true;
                     elsif (doneCount_up and true) then
                        inCount_up <= transport false;
                        inwait_state <= transport true;
                     elsif (doneCount_down and true) then
                        inCount_down <= transport false;
                        inwait_state <= transport true;
                  end if;
                  wait until (not inCount'stable) or (inwait_state and CONSIG(3)
 = '1' and EN and rising_fct(CLK)) or (inwait_state and CONSIG(2) = '1' and EN a
nd rising_fct(CLK)) or (doneCount_up and true) or (doneCount_down and true);
               end process control;
           end block Count;
           Clear: block (inClear and not(inClear'stable))
           begin
               code: process
               begin
               if guard then
               Clear_Loop : loop
               CNT <=  CNT;
               CNT <=  B"0000" after 5 ns;
               wait  until not (inClear) ;
               if (not inClear ) then
               exit Clear_Loop;
               end if;
               exit Clear_Loop;
               end loop Clear_Loop;
               end if;
               CNT <= transport null;
               wait  on guard;
               end process code;
           end block Clear;
           control: process begin
                 if (inCounter and not(inCounter'stable)) then
                    inCount <= transport true;
                 elsif (inCount and CONSIG(0) = '1') then
                    inCount <= transport false;
                    inClear <= transport true;
                 elsif (inClear and not (CONSIG(0) = '1') ) then
                    inClear <= transport false;
                    inCount <= transport true;
              end if;
              wait until (not inCounter'stable) or (inCount and CONSIG(0) = '1')
or (inClear and not (CONSIG(0) = '1') );
          end process control;
       end block Counter;
       Decode: block (inDecode and not(inDecode'stable))
       begin
           code: process
               variable CONREG: bit_vector (1 downto 0);
           begin
           if guard then
           CONSIG <=  CONSIG;
           loop
           wait  on STRB until STRB = '1';
           CONREG := CON;
           case CONREG is
           when "00" =>
```

```
     CONSIG <=  B"0001" after 5 ns;
     when "01" =>
     CONSIG <=  B"0010" after 5 ns;
     when "10" =>
     CONSIG <=  B"0100" after 5 ns;
     when "11" =>
     CONSIG <=  B"1000" after 5 ns;
     end case;
     end loop ;
     wait ;
     end if;
     CONSIG <= transport null;
     wait  on guard;
     end process code;
 end block Decode;
 Load_limit: block (inLoad_limit and not(inLoad_limit'stable))
 begin
     code: process
     begin
     if guard then
     LIM <=  LIM;
     loop
     wait  on STRB until STRB = '0' and CONSIG(1) = '1';
     LIM <=  DATA after 10 ns;
     end loop ;
     wait ;
     end if;
     LIM <= transport null;
     wait  on guard;
     end process code;
 end block Load_limit;
 Update_output: block (inUpdate_output and not(inUpdate_output'stable))
 begin
     code: process
     begin
     if guard then
     CNT_OUT_sig <=  CNT_OUT_sig;
     loop
     CNT_OUT_sig <=  CNT;
     wait  on CNT;
     end loop ;
     wait ;
     end if;
     CNT_OUT_sig <= transport null;
     wait  on guard;
     end process code;
 end block Update_output;
 Update_enable: block (inUpdate_enable and not(inUpdate_enable'stable))
 begin
     code: process
     begin
     if guard then
     loop
     EN <=  CNT /= LIM after 10 ns;
     wait  on LIM,CNT;
     end loop ;
     wait ;
     end if;
     EN <= transport null;
     wait  on guard;
     end process code;
 end block Update_enable;

 control: process begin
       if (inControlled_counter and not(inControlled_counter'stable)) then
          inCounter <= transport true;
          inDecode <= transport true;
          inLoad_limit <= transport true;
```

```
                inUpdate_output <= transport true;
                inUpdate_enable <= transport true;
              end if;
          wait until (not inControlled_counter'stable);
        end process control;
    end block Controlled_counter;

CNT_OUT <= transport CNT_OUT_sig;
start: process begin
  inControlled_counter <= transport true;
  wait;
end process start;

end Controlled_counterA;
```

B DRACO Appendix : SpecChart textual code

State DRACO

```
state
{

name
{
   DRACO
}
declarations
{
   port POWER :  in switch ;
   port CE_L :  in bit ;
   port RESET_L : in  bit ;
   port ALE :  in bit;
   port WRITE_L :  in bit;
   port READ_L :  in bit;
   port ERROR_L :  out bit;
   port PARITY_IN :  in bit;
   port PARITY_OUT:  out bit;
   port AD_IN: in  LSB ;
   port AD_OUT: out  LSB ;
   port MSB_IO_BUS_IN :  in MSB ;
   port MSB_IO_BUS_OUT :  out MSB := X"FF";
   port LSB_IO_BUS_IN :  in LSB ;
   port LSB_IO_BUS_OUT :  out LSB := X"FF";

   signal ADDR_LATCH :  LSB ;
   signal PARITY_LATCH :  bit;
   signal CONFIG_STATUS_REG :  LSB ;
   signal MSB_BUF :  MSB ;
   signal LSB_BUF :  LSB ;
   signal MSB_IO_DIR_REG :  MSB ;
   signal LSB_IO_DIR_REG :  LSB ;
   signal EKEY : key := off;
}
connections
{ }
estimates
{ }
constraints
{ }
sequential substates
{
  WAIT :
    (EI, POWER = onn and not (POWER'stable), POWER_ON),
    (EI, RESET_L='0' and not (RESET_L'stable) and POWER=onn, RESET),
    (EI, (ALE = '0') and not (ALE'stable)
                     and (POWER = onn) and (CE_L = '0'), ADDRESS),
    (EI, (READ_L = '0') and not (READ_L'stable)
                     and (POWER = onn) and (CE_L = '0'), READ),
    (EI, (WRITE_L = '0') and not (WRITE_L'stable)
                     and (POWER = onn) and (CE_L = '0'), WRITE) ;

  POWER_ON :  (EOC, true, WAIT);
  RESET :     (EOC, true, WAIT);
  ADDRESS :   (EOC, true, WAIT);
  READ :      (EOC, true, WAIT);
  WRITE :     (EOC, true, WAIT);
}
}
```

State WAIT

```
state
{
name
{
  WAIT
}
declarations
{ }
connections
{ }
estimates
{ }
constraints
{ }
code
{
     null ;
}
}
```

State POWER_ON

```
state
{

name
{
  POWER_ON
}
declarations
{ }
connections
{ }
estimates
{ }
constraints
{ }
code
{
     ADDR_LATCH <=  transport X"FF";
     MSB_IO_DIR_REG <=  transport X"00";
     LSB_IO_DIR_REG <=  transport X"00";
}
}
```

State RESET

```
state
{

name
{
   RESET
}
declarations
{ }
connections
{ }
estimates
{ }
constraints
{ }
code
{
   ADDR_LATCH <= transport  X"00";
   PARITY_LATCH <= transport  '0';
   CONFIG_STATUS_REG <= transport  X"00";
   MSB_BUF <=  transport X"00";
   LSB_BUF <=  transport X"00";
   MSB_IO_DIR_REG <=  transport X"00";
   LSB_IO_DIR_REG <=  transport X"00";
   ERROR_L <= transport '1';
   EKEY <= off;

}
}
```

State ADDRESS

```
state
{
name
{
  ADDRESS
}
declarations
{ }
connections
{ }
estimates
{ }
constraints
{ }
code
{
  ADDR_LATCH <= transport  AD_IN;
  PARITY_LATCH <= transport  PARITY_IN;
}
}
```

State WRITE

```
state
{

name
{
   WRITE
}
declarations
{
   variable T : time := 120 fs;
   variable TW : time := 60 fs;
   variable MSB_FF : MSB := X"00";
   variable LSB_FF : LSB := X"00";
}
connections
{ }
estimates
{ }
constraints
{ }
code
{
                                                                -- Clear write_ack
  CONFIG_STATUS_REG(4) <= transport  '0';

                                                            -- Check address parity
  if (CONFIG_STATUS_REG(2) = '1') and
     (ODD_PARITY(ADDR_LATCH) /= PARITY_LATCH) then
     assert (false)
     report "ERROR : Parity error in received address, Write aborted"
     severity note;
     ERROR_L <= transport  '0';
     CONFIG_STATUS_REG(3) <= transport  '1';

                                                              -- Check data parity
  elsif ((CONFIG_STATUS_REG(1) = '1') and
         (ODD_PARITY(AD_IN) /= PARITY_IN)) then
     assert (false)
     report "ERROR : Parity error in received data,  Write  aborted"
     severity note;
     ERROR_L <= transport  '0';
     CONFIG_STATUS_REG(3) <= transport  '1';

                                                        -- Write first key address
  elsif (ADDR_LATCH = X"80") then
     if ((AD_IN = X"AA") and (EKEY /= onn)) then
          EKEY <= mid;
     elsif (AD_IN /= X"AA") then
          EKEY <= off;
          CONFIG_STATUS_REG(5) <= '0';
     end if;
                                                       -- Write second key address
  elsif (ADDR_LATCH = X"7F") then
      if (EKEY = onn) then
         if (AD_IN = X"55") then
             CONFIG_STATUS_REG(7 downto 6) <= transport  "01";
         elsif (AD_IN = X"AA") then
             CONFIG_STATUS_REG(7 downto 6) <= transport  "10";
         end if;
      elsif (EKEY = mid) then
         if (AD_IN = X"55") then
           EKEY <= onn;
           CONFIG_STATUS_REG(5) <= '1';
         else
           EKEY <= off;
         end if;
```

```
    end if ;
                                                                  -- Reset Error
  elsif (ADDR_LATCH = X"OF") then
        ERROR_L <= transport  '1';
        CONFIG_STATUS_REG(3) <= transport  '0';

                                                  -- Write High Byte Diection Register
  elsif (ADDR_LATCH = X"04") then
        if (CONFIG_STATUS_REG(7) = '1') then
           MSB_IO_DIR_REG <= transport  AD_IN ;
           MSB_IO_BUS_OUT <= transport MSB_FF or AD_IN after T;
        else assert false
           report "ERROR : Attempt to set MSB IO_DIR with config locked"
           severity note;
           ERROR_L <= transport  '0';
           CONFIG_STATUS_REG(3) <= transport  '1';
        end if;

                                                   -- Write Low Byte Diection Register
 elsif (ADDR_LATCH = X"03") then
        if (CONFIG_STATUS_REG(7) = '1') then
           LSB_IO_DIR_REG <= transport  AD_IN;
           LSB_IO_BUS_OUT <= transport LSB_FF or AD_IN after T;
       else assert false
             report "ERROR : Attempt to set LSB IO_DIR with config locked"
             severity note;
             ERROR_L <= transport  '0';
             CONFIG_STATUS_REG(3) <= transport  '1';
       end if;

                                                       -- Write Configuration Register
 elsif (ADDR_LATCH = X"02") then
     if (CONFIG_STATUS_REG(7) = '1') then
       CONFIG_STATUS_REG (2 downto 0) <= transport AD_IN (2 downto 0);
     else assert false
          report "ERROR :  Attempt to reconfigure DRACO with config locked"
          severity note;
          ERROR_L <= transport  '0';
          CONFIG_STATUS_REG(3) <= transport  '1';
     end if;

                                                         -- the address is 01, 00, OE
elsif ((ADDR_LATCH = X"00") or
       (ADDR_LATCH = X"01") or
       (ADDR_LATCH = X"0E")) then
    if (CONFIG_STATUS_REG(6) = '1') then
       case ADDR_LATCH is
                                                            -- Write to High IO Byte
            when "00000001" =>
                 if (CONFIG_STATUS_REG(0) = '1') then
                    MSB_BUF <= transport  AD_IN;
                 else
                    MSB_BUF <= transport  AD_IN;
                    MSB_FF := ONES_COMP(AD_IN);
                    MSB_IO_BUS_OUT <= transport
                    MSB_FF or MSB_IO_DIR_REG after T;
                    CONFIG_STATUS_REG(4) <= transport  '1';
                 end if;
                 EKEY <= off;
                 CONFIG_STATUS_REG(5) <= transport  '0';
                                                            -- Write to Low IO Byte
            when "00000000" =>
                 if (CONFIG_STATUS_REG(0) = '1') then
                    LSB_BUF <= transport  AD_IN;
                 else
                    LSB_BUF <= transport  AD_IN;
                    LSB_FF := ONES_COMP(AD_IN);
                    LSB_IO_BUS_OUT <= transport
                    LSB_FF or LSB_IO_DIR_REG after T;
                    CONFIG_STATUS_REG(4) <= transport  '1';
                 end if;
                 EKEY <= off;
                 CONFIG_STATUS_REG(5) <= transport  '0';
                                                   -- Write to Checksum IO Byte
            when "00001110" =>
                 if (CONFIG_STATUS_REG(0) = '1') then
                    if (ONES_COMP(LSB_BUF + MSB_BUF) = AD_IN) then
                       MSB_FF := ONES_COMP(MSB_BUF);
                       MSB_IO_BUS_OUT <= transport
                       MSB_FF or MSB_IO_DIR_REG after T;
                       LSB_FF := ONES_COMP(LSB_BUF);
                       LSB_IO_BUS_OUT <= transport
                       LSB_FF or LSB_IO_DIR_REG after T;
                       CONFIG_STATUS_REG(4) <= transport  '1';
                    else assert false
                         report " ERROR : Checksum check failed"
                         severity note;
                         ERROR_L <= transport  '0';
                         CONFIG_STATUS_REG(3) <= transport  '1';
                    end if;
                    EKEY <= off;
                    CONFIG_STATUS_REG(5) <= transport  '0';
                 else assert false
                 report "ERROR :  Checksum Write with checksum option disabled"
                      severity note;
                      ERROR_L <= transport  '0';
                      CONFIG_STATUS_REG(3) <= transport  '1';
                 end if;
            when others =>
                 null;
        end case;
                                                             -- Data is locked
     else assert false
        report "ERROR : Attempt ot write data or checksum with data locked"
        severity note;
        ERROR_L <= transport  '0';
        CONFIG_STATUS_REG(3) <= transport  '1';
     end if;

                                  -- if control reaches here, invalid address
  else assert false
       report "ERROR : Write to an invalid address, Write aborted"
       severity note;
       ERROR_L <= transport  '0';
       CONFIG_STATUS_REG(3) <= transport  '1';
  end if;

  wait  until ((WRITE_L = '1') and not (WRITE_L'stable)
                              and (POWER = onn) and (CE_L = '0'));
}
}
```

State READ

```
state
{
name
{
   READ
}
declarations
{
  variable TRLDV : time := 100 fs;
  variable INTERNAL_DBUS : LSB ;
}
connections
{ }
estimates
{ }
constraints
{ }
code
{

                                                      -- Check Address parity
  if (CONFIG_STATUS_REG(2) = '1') and
     (ODD_PARITY(ADDR_LATCH) /= PARITY_LATCH) then
      assert (false)
      report " ERROR : Parity error in received address, Read aborted"
      severity note;
      ERROR_L <= transport '0';
      CONFIG_STATUS_REG(3) <= transport '1';
      AD_OUT <= transport INTERNAL_DBUS after TRLDV;
      PARITY_OUT <= transport EVEN_PARITY(INTERNAL_DBUS) after TRLDV;

                                                     -- Read inverted checksum
   elsif (ADDR_LATCH = X"0E") then
      INTERNAL_DBUS := ONES_COMP(MSB_BUF + LSB_BUF);
      AD_OUT <= transport INTERNAL_DBUS after TRLDV;
      PARITY_OUT <= transport ODD_PARITY(INTERNAL_DBUS) after TRLDV;

                                              -- Read MSB IO direction register
   elsif (ADDR_LATCH = X"04") then
      INTERNAL_DBUS := MSB_IO_DIR_REG;
      AD_OUT <= transport INTERNAL_DBUS after TRLDV;
      PARITY_OUT <= transport ODD_PARITY(INTERNAL_DBUS) after TRLDV;

                                              -- Read LSB IO direction register
   elsif (ADDR_LATCH = X"03") then
      INTERNAL_DBUS := LSB_IO_DIR_REG;
      AD_OUT <= transport INTERNAL_DBUS after TRLDV;
      PARITY_OUT <= transport ODD_PARITY(INTERNAL_DBUS) after TRLDV;

                                                      -- Read STATUS register
   elsif (ADDR_LATCH = X"02") then
      INTERNAL_DBUS := CONFIG_STATUS_REG;
      AD_OUT <= transport INTERNAL_DBUS after TRLDV;
      PARITY_OUT <= transport ODD_PARITY(INTERNAL_DBUS) after TRLDV;

                                                          -- Read MSB IO bus
   elsif (ADDR_LATCH = X"01") then
      INTERNAL_DBUS := ONES_COMP(MSB_IO_BUS_IN);
      AD_OUT <= transport INTERNAL_DBUS after TRLDV;
      PARITY_OUT <= transport ODD_PARITY(INTERNAL_DBUS) after TRLDV;

                                                          -- Read LSB IO bus
   elsif (ADDR_LATCH = X"00") then
      INTERNAL_DBUS := ONES_COMP(LSB_IO_BUS_IN);
      AD_OUT <= transport INTERNAL_DBUS after TRLDV;
      PARITY_OUT <= transport ODD_PARITY(INTERNAL_DBUS) after TRLDV;
```

```
    else assert false
         report "ERROR : Read attempt from an invalid address, Read aborted"
         severity note;
         ERROR_L <= transport '0';
         CONFIG_STATUS_REG(3) <= transport '1';
         AD_OUT <= transport INTERNAL_DBUS after TRLDV;  -- junk (undefined)
         PARITY_OUT <= transport EVEN_PARITY(INTERNAL_DBUS) after TRLDV;
    end if;

    wait  until ((READ_L = '1') and not (READ_L'stable)
           and (POWER = onn) and (CE_L = '0'));


}
}
```