

UC Irvine

ICS Technical Reports

Title

Benchmarks for the 1992 high level synthesis workshop

Permalink

<https://escholarship.org/uc/item/73b702b6>

Authors

Dutt, Nikil
Ramachandran, Champaka

Publication Date

1992-10-30

Peer reviewed

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

ARCHIVES
Z
699
C3
no. 92-107
c.2

Benchmarks for the 1992 High Level Synthesis Workshop

Nikil Dutt
Champaka Ramachandran

Technical Report #92-107
Oct 30, 1992

University of California, Irvine
Irvine, CA 92717
(714) 856-7219

dutt@ics.uci.edu

Abstract

This report describes the current status of benchmarks for the 1992 High-Level Synthesis Workshop and suggests guidelines for benchmark submission. The benchmark set currently has 9 designs, where each benchmark includes a VHDL description of the design, documentation of the design's functionality, as well as a set of test vectors and expected outputs for simulation. Documentation of the testing strategy the test vectors are also provided with each benchmark. Although the benchmarks are currently written in VHDL, we have attempted to organize the benchmarks in a language-independent format so that users can easily translate the benchmarks into their favorite HDL; the representative set of test vectors and expected outputs allow a user to ensure, with some level of confidence, that their HDL descriptions preserve the original behavior of the benchmarks. The current benchmark set contains designs that exercise different types of functionality (e.g., DSP, FSM-based, arithmetic, etc.), as well as different types of HDL behavioral constructs (e.g., nested loops and nested conditionals). We conclude with a suggested set of guidelines for benchmark submission.

Contents

1	Introduction	1
2	Overview of Benchmarks and Testing Strategy	1
3	Traffic Light Controller	3
4	Armstrong Counter	3
5	Differential Equations	4
6	Elliptic Filter	4
7	Kalman Filter	5
8	Greatest Common Divisor	6
9	Am 2901	6
10	Am 2910	7
11	Intel 8251	8
12	Benchmark Guidelines	8
	12.1 "Well-Known" HDL Description	8
	12.2 Design Documentation, Assumptions, Simplifications	8
	12.3 Simulation Vectors	9
	12.4 Simulator Details	9
	12.5 Synthesis Outputs	9
13	Summary	9
14	Acknowledgments	9
15	References	10
A	Appendix	10

List of Figures

1	Status of HLSW92 Benchmarks	2
----------	--	----------

1 Introduction

The benchmarking effort for High-Level Synthesis (HLS) began during the summer of 1987 when an informal benchmarking discussion was held at the 24th DAC. The urgent need for a set of benchmarks led to the HLSW 1988 Call for Participation stating: *The objective of the workshop is to begin the development of a set of “high-level synthesis benchmarks” that can provide a means of comparing different synthesis systems and guide future work to include a complete range of digital circuits.* This led to the development of an informal set of benchmarks comprising different types of designs such as simple controllers, microprocessors, digital signal processing algorithms and other applications. These benchmarks were subsequently made available through the SIGDA benchmark repository maintained at `mcnc.mcnc.org` under the directories *HLSynth89* and *HLSynth91*.

The old benchmark set was not very robust and had several shortcomings. They lacked documentation of design functionality, and more importantly, generally lacked typical simulation vectors that could be used to verify the “correctness” of the input HDL descriptions, as well as of the synthesized designs.¹ However, we have reached a point of maturity in HLS where several researchers use the benchmarks for comparative evaluation of their results. These comparisons are often confusing and sometimes incorrect, due to the inherent ambiguity in the older set of benchmarks.

In this report, we attempt to rectify this situation by providing a set of sample benchmarks that include design documentation, typical design behaviors described as sample test vectors and expected outputs, and documentation of the testing strategy. The benchmarks are written using a common look-and-feel to maintain consistency across different designs. Although the benchmarks described here have been written in VHDL, we have attempted to organize them in a fairly HDL-independent format so as to facilitate greater usability through ease of translation to other HDLs.

This report concludes with some suggested guidelines for benchmark submission.

2 Overview of Benchmarks and Testing Strategy

This section briefly describes the current set of benchmarks,² which consists of a set of nine designs as summarized in Table 1. These benchmarks vary in the level of design description (e.g., FSM, functional blocks, algorithms), the style of VHDL used as well as in the VHDL control features and data types exercised. Several benchmarks are derived from “familiar” designs used by HLS researchers in the past (e.g., Fifth-order wave elliptic filter and Diffeq). For each design, we have tried to provide documentation of the functionality and assumptions made in coding the VHDL description. The appendix of this report contains a listing of the VHDL behavioral descriptions.

For each benchmark description, we also provide a brief description of the testing strategy used to obtain the set of test patterns for the benchmark, documentation of the test vectors, as well as the actual test patterns and the expected outputs. While these patterns are certainly not exhaustive, we have attempted to provide tests that exercise typical behaviors of the design, with the hope that it will facilitate ease of translation to other HDLs (and other VHDL modeling styles) that are used by individual synthesis tools.

The test patterns for each benchmark can be viewed as “sanity-checks” that attempt to exercise typical behaviors of the benchmark without performing exhaustive testing. As a general testing strategy, we exercise each function of the benchmark, and try to stimulate these functions under

¹A notable exception was the set of Hardware-C descriptions that included sample test vectors and expected outputs.

²These benchmarks are available by anonymous ftp from `mcnc.mcnc.org` under `pub/benchmark/HLSynth92` and from `ics.uci.edu` under `pub/HLSynth92`.

Figure 1: Status of HLSW92 Benchmarks

Design name	Design description	Design level	VHDL description style	Control feature	Data types	Test vectors
Traffic light controller	FSM	FSM beh.	1 Process	Nested Case, nested Ifs	Bit Vectors	23
Diffeq	Differential equation Solver	Algorithmic Beh.	1 Process	Embedded Loop, straight line code	Integers	12
Kalman filter	Digital Filter	Algorithmic Beh.	1 Process	Nested Loops, nested Ifs	Signed Integers, Arrays	8
Armstrong counter	Controlled Counter	Functional & Algorithmic Beh.	4 Process	Nested Ifs	Bit Vectors	24
Intel 8251	USART	Functional & Algorithmic Beh.	3 Process, 2 blocks	Nested Ifs, nested Case, nested Loops	Bit Vectors	> 20000
Am2901	Microprocessor Slice	Algorithmic Beh.	(a) 1 Process (b). 5 blocks	Case, Ifs	Bit Vectors	216
Am2910	Microprogram Address Sequencer	Functional & Algorithmic Beh.	(a) 1 Process (b) 5 blocks	Nested Ifs, Case	Bit Vectors	635
Ellipf	Digital Filter	Algorithmic Beh.	1 Process	straight line code	Bit Vectors	6
GCD	GCD Algorithm	Algorithmic Beh.	1 Process	Ifs Nested in loop	Bit Vectors	24

different combinations of inputs. For designs that are partitioned into components, we attempt to test each component in different modes with test vectors designed to detect "stuck-at-0" and "stuck-at-1" faults at various points in the hardware. The paths are also tested for "stuck-at-0" and "stuck-at-1" faults at any point on the path. In addition, we try to test some specified ports in both their complimentary forms (1 and 0) which is analogous to testing for "stuck-at" faults in the synthesized hardware.

3 Traffic Light Controller

Description

This benchmark describes a traffic light controller that regulates traffic at the intersection of a highway and a sideroad. The model is written such that the highway has priority over the side road. It uses a short timeout and a long timeout along with the traffic data on the side road to determine the length of time the traffic lights are in a particular state. The detailed functioning is described in [MeCo80].

Assumptions

The model assumes that some clocking signal is available to generate the timeout signals.

Testing Strategy

The tests consists of exercising paths to encompass all sequence of events. These tests ensure that all sequences of events behave in the predicted manner.

4 Armstrong Counter

Description

The Armstrong counter counts up or down till a prespecified external limit is reached. It operates with the Clock and Strobe signals acting as triggers.

The counting is done on the positive edge of clock. The decoding is performed on the positive edge of the strobe signal and the limit is loaded on a negative edge of Strobe signal.

If the counter reaches the limit, further counting is disabled till a new limit is loaded or the counter is cleared.

The benchmark was derived from the controlled counter description in [Arms89]

Caveats

The model does not support the behavior of the counter under these conditions:

- While the counter is counting towards the limit, the counting direction is changed. This means the counting should stop since the value now has crossed the limit in the direction of count;

however, the model does not support this behavior.

For example, let the counter's state be at 1 and counting-up, with the limit set at 7. If the count direction is now changed to downward count, the counter has already crossed the limit 7 during count down, so it should stop counting.

- While the counter is counting towards the limit, the limit is changed so that the counter now exceeds the limit. This means the counting should stop since the value now has crossed the limit in the direction of count; however, the model does not support this behavior.

For example, let the counter's state be at 7 and counting-up with the limit set at 14. If the limit is now changed to 5, the counter has already crossed the limit 5 during count down, so it should stop counting.

Testing Strategy

The testing strategy consists of running a clock process and a counter testing process. The types of tests include performing a complete count-up and count-down sequence and also testing whether the limit function works while performing the count-up and count-down.

5 Differential Equations

Description

This benchmark provides the hardware description for a small fixed-point calculation loop. The algorithm tries to numerically solve the equation

$$y'' + 3xy' + 3y = 0$$

Here, u is assumed to represent dy/dx or y' . dx is approximated as $x1 - x$. Similarly, $dy = y1 - y$ and $du = u1 - u$. The value 'a' provides the number of times the numerical loop is executed. $u1$, $x1$ and $y1$ represent the new values of u , x and y . Thus, $x1 = x + dx$, $y1 = udx + y$, $u1 = u - 3xudx - 3ydx$. The behavior executes by loading the initial values of x , y , u , dx , and a .

This benchmark was derived from [BrGa87].

Testing Strategy

For this benchmark, the tests include checking the execution of the loop a desired number of times and checking for overflow on the outputs. We also check for correct operation under different conditions such as when the inputs are zeroes or negative numbers.

6 Elliptic Filter

Descriptions

The elliptic filter belongs to the class of Infinite Impulse Response (IIR) filters, where the filter's response to an impulse input remains non-zero till infinite time in a theoretical sense. The particular

filter we deal with here is a low pass filter, meaning that it filters off frequencies higher than a certain limit, called the cut-off frequency.

This filter design description is composed of a basic block of several arithmetic operations, and has long been a popular benchmark for comparing the results of scheduling. The benchmark derived from descriptions in [KuWK85] and [Orch90].

Testing Strategy

The functional testing of the elliptic filter is usually done with a 'delta' function, the rough approximation of which in the digital domain is a vector that is '1' in the first instance, and is '0' for all other states. The corresponding output has to be tested by substituting the vector in the z-transform of the state equation.

The vectors were derived heuristically, according to standard practices used in such cases. We consider vectors that are all similar (say all 0's and all 1's), vectors that have different combinations of even and odd numbers as current state vectors, and vectors that are powers of 2.

7 Kalman Filter

Description

The purpose of the Kalman filter is to predict the state vector of a system from a set of observed quantities. The dimensionality of the state vector is larger than that of the measurements. The imbalance is remedied by using many successive data observations in the prediction process.

The operation of the Kalman filter chip is as follows: First a set of coefficient matrices is downloaded into the chip. Once this is completed, the chip enters its control loop. Within this loop, four steps are repeated indefinitely. First, 13 measurements "y" are read into the chip. Second, the state vector "x" (of dimension 16) is estimated. This involves multiplication by a 16x16 matrix, multiplication by a 16x13 matrix, and numerical integration using the previous state estimate. Third, the control output vector "v" (of dimension 4) is computed from the state estimate. This involves multiplication by a 4x16 matrix. Fourth, the control vector "v" is output from the chip.

The Kalman filter model was derived from [Newt92].

Assumptions

The Kalman filter requires negative numbers to control its feedback. The highest negative number is two to the fifteen. Also, coefficients below unity are required to ensure that feedback does not cause the numbers to blow up. Hence, all inputs numbers have been multiplied by 1024, that is pitched 10 binary places above unity.

Testing Strategy

We start off the testing process by loading the constant matrices A, G and K. There exists 2 types of test sequences. Four different run of test vectors was followed by a run of four identical inputs. This is done to demonstrate the convergence of the state vector estimation process. We have also included some sanity check vectors.

Complete details of the testing strategy can be obtained in [Newt92].

8 Greatest Common Divisor

Description

The gcd model described in this example consists of two 8-bit input ports, one 8-bit output port, and a one-bit input port that enables the gcd model when the value is low.

The gcd model can be enabled by setting `rst='0'`. The output `ou` will be evaluated as `gcd(xi, yi)` based on the input values of `xi` and `yi`. If `rst='1'`, then the output will be evaluated to be "00000000".

This algorithm is derived from [BrBr].

Testing Strategy

All possible paths in this benchmark are tested. The test types include checking when the input numbers are multiples, and when the numbers are not multiples of each other. The other sets of tests include the case when the input numbers are large.

9 Am 2901

Description

Am2901 is a four-bit microprocessor slice (from Advanced Micro Devices Inc.) It can be described either using functional blocks or by its behavior.

Its main functional blocks are as follows :

- 16-word by four bit two port RAM, with an up/down shifter at the input.
- A register (called Q) with an up/down shifter at the input
- An ALU source selector which select two inputs out of, Port A of RAM, Port B of RAM, Q register output, External data input and Logical 0
- A 4-bit ALU, capable of performing arithmetic and logical functions on the selected source words.
- A destination selector which decides whether to load the ALU output (with or without shifting) into the RAM, whether to load the ALU output (with or without shifting) or the Q register contents (with shifting) into the Q register OR whether the ALU output or the Port A contents should be forwarded to the External data output.

The behavior description of 2901 consists of a VHDL process that has three case statements corresponding to ALU operand selection, ALU function selection and ALU destination and data-output selection.

In the model, data is first read into "A" and "B" from the RAM words addressed by `Aadd` and `Badd`. Then the ALU operands are selected. The ALU does computation on these operands. After

that, the destination selector decides whether and how to write the ALU results to the RAM and Q register.

The complete details of the function of this design can be found in [AMDe82]. Further details of the model can be found in [Ghos92].

Testing Strategy

There are two types of paths in this design. One path starts at some register (or external data input), goes through the ALU and ends at a register. The other path starts at a register, goes via the ALU and ends at a RAM. Each of these paths tests a different ALU source line.

Further details of the test patterns can be found in [Ghos92].

10 Am 2910

Description

The Am2910 is a microprogram address sequencer intended for use in high-speed microprocessor applications [AMDe89].

The Am2910 has a four-input multiplexor that is used to select either the register/counter (R), direct data input (D), microprogram counter (uPC) or the top of stack (TOS) as the source of the next microinstruction address.

The register/counter performs the operations of load or decrement. The microprogram counter is used when incrementing needs to be performed, to execute sequential microinstructions. The third source for the multiplexor is the direct (D) input. This source is used for branching. The fourth source available at the multiplexor input is the top of the stack which is used to provide return address linkage when executing microsubroutines or loops.

The device provides three-state Y outputs. These can be particularly useful in designs requiring automatic checkout of the processor. The microprogram sequencer outputs can be forced into high-impedance state, and pre-programmed sequences of microinstructions can be executed via external access to the address lines.

The detailed model is described in [Ghos92].

Testing Strategy

In testing the Am2910 models, the overall strategy adopted is to test each "hardware" component (e.g. stack, register/counter etc.) using sequences of test vectors.

Because the components are not being tested in isolation, we need to set up input values at input ports of the chip and propagate them to the input of that component. Also, the output of a component has to be propagated to the output ports of the chip.

Further details of the test patterns can be obtained in [Ghos92].

11 Intel 8251

Description

The Intel 8251 Universal Synchronous/Asynchronous Receiver/Transmitter (USART), designed for data communication with Intel's microprocessor families described in [Inte81]. It is used as a peripheral device and is programmed by the CPU to operate using many serial data transmission techniques.

The USART accepts data characters from the CPU in parallel format and then converts them into a continuous serial data stream. It accepts serial data streams and converts them into parallel data characters for the CPU. The USART will signal the CPU whenever it can accept a new character for transmission or whenever it has received a character for the CPU. The CPU can read the status of the USART at any time.

The complete functional definition of the 8251 is programmed by the system's software. A set of control words must be sent out by the CPU to initialize the 8251 to support the desired communication format. These words must immediately follow a reset (internal/external).

The VHDL model consists of three major processes "main", "receiver" and "transmitter". The model describes how each of the above process handle the various mode words namely, Synchronous mode word, Asynchronous mode word Command word, and Status Word.

It also describes the operation in the following modes, Asynchronous Mode (Transmission), Synchronous Mode (Transmission), Asynchronous Mode (Receive) and the Synchronous Mode (Receive).

Further details regarding the model can be obtained from [Ghos92].

Testing Strategy

In testing the functionality of the 8251, we mainly concentrate on testing its main operational modes, Synchronous transmission, Asynchronous transmission Synchronous receive (External Synchronization), Synchronous receive (Internal Synchronization) and Asynchronous receive.

Further details regarding the test patterns can be obtained from [Ghos92].

12 Benchmark Guidelines

In this section, we suggest some guidelines for the submission of new benchmarks. This is a first step towards introducing more rigor in the benchmarking process, and towards the creation of a robust set of benchmark examples for testing High-Level Synthesis tools and systems.

12.1 "Well-Known" HDL Description

The design must be described using a "well-known" HDL which has a publicly available LRM, and which has a publicly available simulator. Sample HDLs that fit this criterion include VHDL, Verilog and Hardware-C.

The HDL description must be liberally commented to allow readability.

12.2 Design Documentation, Assumptions, Simplifications

The source of the design information should be specified (e.g., data sheet, initial design spec., etc.).

A description of the design's functionality (using English, flowcharts, block diagrams, etc.) must accompany the HDL description.

All assumptions and simplifications made in writing the HDL model must be clearly stated.

12.3 Simulation Vectors

A set of input and expected output functional test vectors must accompany the HDL description for simulating typical operational behaviors of the design. These test vectors are not designed to exhaustively test the design. Instead, they give some level of confidence in the behavioral HDL model, and allow translation and validation of the model into another HDL or description style.

The test vectors must also be accompanied by a (English) description of what functionality is being tested.

The input and expected output vectors should be described in a generic format that allows ease of use in different simulation environments. A brief description of the test vector format must accompany the test vector set.

12.4 Simulator Details

Each benchmark design must indicate the name, version, and availability (where appropriate) of the simulator used to test the design.

12.5 Synthesis Outputs

The outputs of synthesis tools must be simulated using the same simulator and test vectors used to check the behavior of the input description.

13 Summary

This report presented the status of, and briefly described the benchmark set developed for the Sixth International Workshop on High-Level Synthesis. Several researchers are in the process of contributing more benchmarks; these will be placed in the HLSW92 benchmark repositories both at MCNC and at U.C. Irvine, as soon as they are complete. We will periodically provide updates on the status of benchmarks through the High-Level Synthesis Workshop electronic mailing list.

We are actively soliciting (new or old) benchmarks that follow the suggested guidelines, and ask that you help us create a more comprehensive set of benchmarks by providing design examples.

In conclusion, it should be noted that this is still a preliminary effort in standardizing the benchmarks for High-Level Synthesis. We have yet to resolve several difficult issues, including a standard mechanism for specifying timing and other constraints in the test data sets. We look forward to receiving feedback, comments, suggestions and criticisms.

14 Acknowledgments

We would to thank the following people for their help in benchmark preparation: Indraneel Ghosh (Am2901, Am2910, I8251), Ted Lee (Greatest Common Divisor), D. Sreenivasa Rao (Elliptic Filter) and Joe Lis (Differential Equations, Armstrong Counter). We are grateful to Prof. Daniel Gajski for

his constant encouragement, support and suggestions in this effort. We would also like to thank Prof. Fadi Kurdahi for his support of this activity. This work was supported in part by NSF grants MIP 9009239 and MIP 8922851.

15 References

- [AMDe82] Advanced Micro Devices, Inc, "Am2901 Four-Bit Bipolar Microprocessor Slice," 1982.
- [AMDe89] Advanced Micro Devices, Inc, "Am2910 Microprogram Controller," 1989.
- [Arms89] James Armstrong, "Chip-level Modeling with VHDL," Prentice Hall 1989.
- [BrBr] Gilles Brassard and Paul Bratley, "Algorithmics Theory and Practice" Prentice Hall 1988.
- [BrGa87] F.D.Brewer and D.D Gajski, "Knowledge Based Control in Micro-Architecture Design," Proceedings of 24th DAC, 1987.
- [Ghos92] Indraneel Ghosh, "High-level Modeling of Standard Parts in VHDL," M.S Thesis, Dept. of Electrical and Computer Engg., University of California at Irvine, June 1992.
- [Inte81] Intel Corporation, "Peripheral Design Handbook," 1981.
- [KuWK85] S.Y. Kung, H.J. Whitehouse and T. Kailath, "VLSI and Modern Digital Signal Processing," Prentice Hall 1985, pp. 258-264.
- [MeCo80] Carver Mead and Lynn Conway, "Introduction to VLSI Systems," Addison-Wesley 1980.
- [Newt92] Cleland Newton, "A Synthesis Process Applied to the Kalman Filter Benchmark," Manuscript provided by Cleland Newton, DRA Malvern, UK.
- [Orch90] H. J. Orchard, "Adjusting the Parameters in Elliptic-Function Filters," IEEE Trans CAS, vol 37, no 5, May 1990.

A Appendix

The VHDL models for all the benchmarks are shown below.

```

-----
-- Traffic Light Controller (TLC)
-- Source: Hardware C version written by David Ku on June 8, 1988 at Stanford
-- VHDL Benchmark author: Champaka Ramachandran
--                           University of California, Irvine, CA 92717
--                           champaku@balboa.eng.uci.edu
--
-- Developed on Aug 11, 1992
-- Verification Information:
--
--           Verified   By whom?       Date       Simulator
--           -----   -
-- Syntax          yes    Champaka Ramachandran Aug 11, 92  ZYCAD
-- Functionality   yes    Champaka Ramachandran Aug 11, 92  ZYCAD
-----

```

```

entity TLC is
  port (
    Cars : in BIT;
    TimeoutL : in BIT;
    Timeouts : in BIT;
    StartTimer : out BIT;
    HiWay : out BIT_VECTOR(2 downto 0);
    FarmL : out BIT_VECTOR(2 downto 0);
    state : out BIT_VECTOR(2 downto 0) := '111'
  );
end TLC;

architecture TLC of TLC is

begin
-----
  traffic:process

    variable newstate, current_state : BIT_VECTOR(2 downto 0) := '111';
    variable newHL, newPL : BIT_VECTOR(2 downto 0);
    variable newST : BIT;

  begin
    current_state := newstate;

  -- combinational logic to determine nextstate
  case current_state is

    when '000' => newHL := '100'; newPL := '110';
      if (Cars = '1') and (TimeoutL = '1') then
        newstate := '100'; newST := '1';
      else
        newstate := '000'; newST := '0';
      end if;

    when '100' => newHL := '010'; newPL := '110';
      if (Timeouts = '1') then
        newstate := '010'; newST := '1';
      else
        newstate := '110'; newST := '0';
      end if;

    when '010' => newHL := '110'; newPL := '100';
      if (Cars = '0') or (TimeoutL = '1') then
        newstate := '110'; newST := '1';
      else
        newstate := '010'; newST := '0';
      end if;

    when '110' => newHL := '110'; newPL := '010';
      if (Timeouts = '1') then
        newstate := '000'; newST := '1';
      else
        newstate := '110'; newST := '0';
      end if;

    when '111' =>
      newstate := '000';
      newHL := '000';
      newPL := '000';
      newST := '0';

    when others =>
      end case;

    state <= newstate;
    HiWay <= newHL;
    FarmL <= newPL;
    StartTimer <= newST;
    wait for 10 ns;

  end process traffic ;
-----
end TLC;

```

```

-----
-- Controlled Counter Benchmark
-- Source: *Chip Level Modeling with VHDL* by Jim Armstrong (Prentice-Hall 1989)
-- Benchmark author: Joe Lis
-- Copyright (c) by Joe Lis 1988
-- Modified by : Champaka Ramachandran on Aug 24th 1992
-- Verification Information:
--
--          Verified  By whom?          Date          Simulator
--          -----  -
-- Syntax      yes     Champaka Ramachandran  24/8/92     XYCAD
-- Functionality yes     Champaka Ramachandran  24/8/92     XYCAD
-----

use work.BIT_FUNCTIONS.all;

entity ARMS_COUNTER is
  port (
    CLK: in BIT;
    STRB: in bit;
    CON: in BIT_VECTOR(1 downto 0);
    DATA: in BIT_VECTOR(3 downto 0);
    COUNT: out BIT_VECTOR(3 downto 0));
end ARMS_COUNTER;

--VSS: design_style behavioural

architecture ARMS_COUNTER of ARMS_COUNTER is
  signal ENIT, RENIT: BIT;
  signal EN: BIT;
  signal CONSIG, LIM: BIT_VECTOR(3 downto 0);
  signal CNT : BIT_VECTOR(3 downto 0);

begin
  ----- The decoder -----
  DECODE: process (STRB, RENIT)
  variable CONREG: BIT_VECTOR(1 downto 0) := "00";
  begin
    if (STRB = '1') and (not STRB'STABLE) then
      CONREG := CON;
      case CONREG is
        when "00" => CONSIG <= "0001";
        when "01" => CONSIG <= "0010";
        when "10" => CONSIG <= "0100"; ENIT <= '1';
        when "11" => CONSIG <= "1000"; ENIT <= '1';
        when others =>
          end case;
      end if; -- Rising edge of STRB

      if (RENIT = '1') and (not RENIT'STABLE) then
        ENIT <= '0';
      end if;
    end process DECODE;

  ----- The limit loader -----
  LOAD_LIMIT: process (STRB)
  begin
    if (CONSIG(1) = '1') and (not STRB'STABLE) and (STRB = '0') then
      LIM <= DATA;
    end if;
  end process LOAD_LIMIT;

  ----- The counter -----
  CTR: process (CONSIG(0), EN, CLK)
  variable CNTK : BIT := '0';
  begin
    if (CONSIG(0) = '1') and (not CONSIG(0)'STABLE) then
      CNT <= "0000";
    end if;

    if (not EN'STABLE) then
      if (EN = '1') then
        CNTK := '1';
      else
        CNTK := '0';
      end if;
    end if;

    if (not CLK'STABLE) and (CLK = '1') and (CNTK = '1') then
      if (CONSIG(2) = '1') then
        CNT <= CNTK + "0001";
      elsif (CONSIG(3) = '1') then
        CNT <= CNT - "0001";
      end if;
    end if;
  end process CTR;

  ----- The comparator -----
  LIMIT_CHK: process (CNT, ENIT)
  begin
    if (not ENIT'STABLE) then
      if (ENIT = '1') then
        EN <= '1'; RENIT <= '1';
      else
        RENIT <= '0';
      end if;
    end if;

    if (EN = '1') and (CNT = LIM) then
      EN <= '0';
    end if;
  end process LIMIT_CHK;
end process LIMIT_CHK;
COUNT <= CNT;
end ARMS_COUNTER;

```

```

-----
-- Differential Equation Benchmark
--
-- Source: Adapted from example in paper
--         "HAL: A Multi-Paradigm Approach to Automatic Data Path Synthesis"
--         by P. Paulin, J. Knight and E. Girozyc
--         23rd DAC, June 1986, pp. 263-270
--
-- Benchmark author: Joe Lis
--
-- Copyright (c) 1989 by Joe Lis
--
-- Modified by Champaka Ramachandran on Aug 17th 1992
--
-- Verification Information:
--
--           Verified      By whom?      Date      Simulator
--           -----      -
-- Syntax          yes      Champaka Ramachandran  17th Aug 92  ZYCAD
-- Functionality   yes      Champaku Ramuchandran  17th Aug 92  ZYCAD
-----

```

```

entity diffeq is
  port (Xinport: in integer;
        Xoutport: out integer;
        DXport: in integer;
        Aport: in integer;
        Yinport: in integer;
        Youtport: out integer;
        Uinport: in integer;
        Uoutport: out integer);
end diffeq;

--VSS: design_style BEHAVIORAL;

architecture diffeq of diffeq is
begin
  P1 : process (Aport, DXport, Xinport, Yinport, Uinport)
    variable x_var,y_var,u_var, a_var, dx_var: integer ;
    variable x1, y1, t1,t2,L3,L4,L5,L6: integer ;
  begin
    x_var := Xinport;  u_var := Aport; dx_var := DXport; y_var := Yinport; u_var :=
    while (x_var < a_var) loop
      t1 := u_var * dx_var;
      t2 := 3 * x_var;
      t3 := 3 * y_var;
      t4 := t1 * t2;
      t5 := dx_var * t3;
      t6 := u_var - t4;

      u_var := t6 - t5;
      y1 := u_var * dx_var;
      y_var := y_var + y1;
      x_var := x_var + dx_var;
    end loop;

    Xoutport <= x_var;
    Youtport <= y_var;
    Uoutport <= u_var;
  end process P1;
end diffeq;

```

```

-----
--
-- Elliptical Wave Filter Benchmark
--
-- VHDL Benchmark author: D. Sreenivasa Rao
-- University Of California, Irvine, CA 92717
-- dsr@balboa.eng.uci.edu, (714)856-5106
--
-- Developed on 12 September, 1992
--
-- Verification Information:
--
--
-- Verified      By whom?      Date      Simulator
-- -----
-- Syntax        yes           DSR       09/12/92  %YCAD)
-- Functionality yes           DSR       09/12/92  %YCAD)
-----
--use std_logic.all;
use work.bit_functions.all;

entity ellipf is
  port ( inp : in BIT_VECTOR(15 downto 0);
        outp : out BIT_VECTOR(15 downto 0);
        sv2, sv13, sv18, sv26, sv33, sv38, sv39 :
          in BIT_VECTOR(15 downto 0);
        sv2_o, sv13_o, sv18_o, sv26_o, sv33_o, sv38_o, sv39_o :
          out bit_vector(15 downto 0));
end ellipf;

architecture ellipf of ellipf is
begin
  process (inp, sv2, sv13, sv18, sv26, sv33, sv38, sv39)
-- constant m1, m2, m3, m4, m5, m6, m7, m8 : integer := (1,1,1,1,1,1,1,1);
variable n1, n2, n3, n4, n5, n6, n7 : BIT_VECTOR(15 downto 0);
variable n8, n9, n10, n11, n12, n13 : BIT_VECTOR(15 downto 0);
variable n14, n15, n16, n17, n18, n19 : BIT_VECTOR(15 downto 0);
variable n20, n21, n22, n23, n24, n25 : BIT_VECTOR(15 downto 0);
variable n26, n27, n28, n29 : BIT_VECTOR(15 downto 0);
-- constant i : integer := (1);
begin
-- while (i = 1) LOOP
n1 := inp + sv2;
n2 := sv33 + sv39;
n3 := n1 + sv13;
n4 := n3 + sv26;
n5 := n4 + n2;
n6 := n5 ;
n7 := n5 ;
n8 := n3 + n6;
n9 := n7 + n2;
n10 := n3 + n8;
n11 := n8 + n5;
n12 := n2 + n9;
n13 := n10 ;
n14 := n12 ;
n15 := n1 + n13;
n16 := n14 + sv39;
n17 := n1 + n15;
n18 := n15 + n8;

n19 := n9 + n16;
n20 := n16 + sv39;
n21 := n17 ;
n22 := n18 + sv18;
n23 := sv38 + n19;
n24 := n20 ;
n25 := inp + n21;
n26 := n22 + n21;
n27 := n23 ;
n28 := n26 + sv18;
n29 := n27 + sv38;
sv2_o <= n25 + n15;
sv13_o <= n17 + n28;
sv18_o <= n28;
sv26_o <= n9 + n11;
sv38_o <= n29;
sv33_o <= n19 + n29;
sv39_o <= n16 + n24;
-- outp <= n24;
-- end LOOP;
end process;
end ellipf;

--configuration ellipcon of ellipf is
-- for ellipf_beh
-- end for;
--end ellipcon;

```



```

-----
-- GCD factorization Benchmark
--
-- Source: "Algorithms by Brassard and Bradley"
-- VHDL Benchmark author: Champaka Ramachandran on Sept 11 1992
-- Verification Information:
--
--           Verified      By whom?      Date      Simulator
--           -----      -
-- Syntax      yes      Champaka Ramachandran  11th Sept 92  ZYCAD
-- Functionality yes      Champaka Ramachandran  11th Sept 92  ZYCAD
-----
use work.II1_FUNCTIONS.all;

entity GCD is
  port (X, Y      : in bit_vector(7 downto 0);
        Reset     : in bit;
        gcd_output : out bit_vector(7 downto 0));
end GCD;

architecture GCD of GCD is
begin
  process(X, Y, Reset)

  variable xvar,yvar : bit_vector (7 downto 0);
  variable resetvar  : bit;
  variable compare_var : bit_vector (1 downto 0);

  begin

    xvar := X;
    yvar := Y;
    resetvar := Reset;

    if (xvar = "00000000") then
      gcd_output <= "00000000";
    end if;

    if (yvar = "00000000") then
      gcd_output <= "00000000";
    end if;

    -- The GCD factorization takes place only if Reset = 0
    if (resetvar = '0') and (xvar /= "00000000") and (yvar /= "00000000") then
      compare_var := COMPARE(xvar, yvar);

      -- If compare returns 11 then inputs are equal
      -- If compare returns 10 then xvar > yvar
      -- If compare returns 01 then xvar < yvar

      while (compare_var /= "11") loop

      -- loop till the numbers are equal

        if (compare_var = "01") then
          yvar := yvar - xvar;
        else
          xvar := xvar - yvar;
        end if;

        compare_var := COMPARE(xvar, yvar);
      end loop;

      gcd_output <= xvar;

    else
      gcd_output <= "00000000";
    end if;

  end process;
end GCD;

```

```

-----
-- AM2901 Benchmark
-- Source: AMI data book
-- VHDL Benchmark author: Indraneel Ghosh
--                           University of California, Irvine, CA 92717
-- Developed on Jan 1, 1992
-- Verification Information:
--
-- Verified      By whom?      Date      Simulator
-------
-- Syntax        yes  Champaka Ramachandran  Sept19, 92  ZYCAD
-- Functionality  yes  Champaka Ramachandran  Sept19, 92  ZYCAD
-----

use work.TYPES.all;
use work.MVI7_functions.all; -- some MVI7 functions
use work.synthesis_types.all; -- some data types ( hints for synthesis)

entity a2901 is
  port (
    I : in MVI7_vector(8 downto 0);
    Aadd, Badd : in integer range 0 to 15;
    D : in MVL7_vector(3 downto 0);
    Y : out MVL7_vector(3 downto 0);
    RAM0, RAM3, Q0, Q3 : inout MVI7;
    CLK : in clock;
    CO : in MVL7;
    OEBar : in MVL7;
    C4, Gbar, Pbar, OVH, F3, F30 : out MVI7
  );
end a2901;

architecture a2901 of a2901 is
begin
  process

    variable A, B : MVI7_vector(3 downto 0);
    variable RAM : Memory(15 downto 0);
    variable Q : MVL7_vector(3 downto 0);
    variable RE, S : MVL7_vector(3 downto 0);
    variable F : MVI7_vector(3 downto 0);
    variable dout : MVI7_vector(3 downto 0);
    variable R_ext, S_ext, result : MVL7_vector(4 downto 0);
    variable temp_p, temp_g : MVL7_vector(3 downto 0);

  begin

    wait until ( clk = '0') and (not clk'stable) );

    A := RAM(Aadd); -- RAM OUTPUTS ( ADDRESSED BY Aadd AND Badd ) AND
    B := RAM(Badd); -- MADE AVAILABLE TO ALU SOURCE SELECTOR

    -- SELECT THE SOURCE OPERANDS FOR ALU. SELECTED OPERANDS ARE *RE* AND *S*.

    case I(2 downto 0) is
      when "000" =>
        RE := A;
        S := Q;
      when "001" =>
        RE := A;
        S := B;
      when "010" =>
        RE := "0000";
        S := Q;
      when "011" =>
        RK := "0000";
        S := B;
      when "100" =>
        RK := "0000";
        S := A;
      when "101" =>
        RE := D;
        S := A;
      when "110" =>
        RK := D;
        S := Q;
      when "111" =>
        RK := D;
        S := "0000";
      when others =>
        RE := A;
        S := "0000";
    end case;

    -- SELECT THE FUNCTION FOR ALU.

    -- TO FACILITATE COMPUTATION OF CARRY-OUT *C4*, WE EXTEND THE CHOSEN
    -- ALU OPERANDS *RE* AND *S* (4 BIT OPERANDS) BY 1 BIT IN THE MSB POSITION.

    -- THUS THE EXTENDED OPERANDS *R_ext* AND *S_ext* (5 BIT OPERANDS) ARE
    -- FORMED AND ARE USED IN THE ALU OPERATION. THE EXTRA BIT IS SET TO '0'
    -- INITIALLY. THE ALU'S EXTENDED OUTPUT ( 5 BITS LONG) IS *result*.

    -- IN THE ADD/SUBTRACT OPERATIONS, THE CARRY-INPUT *CO* (1 BIT) IS EXTENDED
    -- BY 4 BITS ( ALL '0') IN THE MORE SIGNIFICANT BITS TO MATCH ITS LENGTH TO
    -- THAT OF *R_ext* AND *S_ext*. THEN, THESE THREE OPERANDS ARE ADDED.

    -- ADD/SUBTRACT OPERATIONS ARE DONE ON 2'S COMPLEMENT OPERANDS.

    case I(5 downto 3) is
      when "000" =>
        R_ext := '0' & RK;
        S_ext := '0' & S;
        result := R_ext + S_ext + ('0000' & CO);
      when "001" =>
        R_ext := '0' & not(RK);
        S_ext := '0' & S;
        result := R_ext + S_ext + ('0000' & CO);
      when "010" =>
        R_ext := '0' & RK;
        S_ext := '0' & not(S);
        result := R_ext + S_ext + ('0000' & CO);
      when "011" =>
        R_ext := '0' & RE;
        S_ext := '0' & S;
        result := R_ext or S_ext;
      when "100" =>
        R_ext := '0' & RE;
        S_ext := '0' & S;
        result := R_ext and S_ext;
      when "101" =>
        R_ext := '0' & RE;
        S_ext := '0' & S;
        result := not(R_ext) and S_ext;
      when "110" =>
        R_ext := '0' & RE;
        S_ext := '0' & S;
        result := R_ext xor S_ext;
    end case;
  end process;
end a2901;

```

```

when '111' =>
  R_ext := '0' & RE;
  S_ext := '0' & S;
  result := not(R_ext xor S_ext);
when others =>
  end case;

-- EVALUATE OTHER ALU OUTPUTS.

-- FROM EXTENDED OUTPUT *result* ( 5 BITS), WE OBTAIN THE NORMAL ALU OUTPUT,
-- *F* (4 BITS) BY LEAVING OUT THE MSB ( WHICH CORRESPONDS TO CARRY-OUT
-- *C4*).

-- TO FACILITATE COMPUTATION OF CARRY LOOKAHEAD TERMS *Pbar* AND *Gbar*, WE
-- COMPUTE INTERMEDIATE TERMS *temp_p* AND *temp_g*.

C4 <= result(4);
OVR <= not (R_ext(3) xor S_ext(3)) and
      (R_ext(3) xor result(3));
F := result(3 downto 0);
temp_p := R_ext(3 downto 0) or S_ext(3 downto 0);
temp_g := R_ext(3 downto 0) and S_ext(3 downto 0);
Pbar <= not( temp_p(0) and temp_p(1) and temp_p(2) and temp_p(3) );
Gbar <= not( temp_g(3) or
      (temp_p(3) and temp_g(2) ) or
      (temp_p(3) and temp_p(2) and temp_g(1) ) or
      (temp_p(3) and temp_p(2) and temp_p(1) and temp_g(0) )
      );
F3 <= result(3);
F30 <= not( result(3) or result(2) or result(1) or result(0) );

-- GENERATE INTERMEDIATE OUTPUT *dout* AND BIDIRECTIONAL SHIFTER SIGNALS.

-- WRITE TO DESTINATION(S) WITH/WITHOUT SHIFTING. RAM DESTINATIONS ARE
-- ADDRESSED BY *Badd*.

case I(8 downto 6) is
  when "000" =>
    dout := F; -- INTERMEDIATE OUTPUT
    Q := F; -- WRITE TO DESTINATION
    Q0 <= 'Z';
    Q3 <= 'Z';
    RAM0 <= 'Z';
    RAM3 <= 'Z';
  when "001" =>
    dout := F;
    Q0 <= 'Z';
    Q3 <= 'Z';
    RAM0 <= 'Z';
    RAM3 <= 'Z';
  when "010" =>
    dout := A;
    RAM(Badd) := F;
    Q0 <= 'Z';
    Q3 <= 'Z';
    RAM0 <= 'Z';
    RAM3 <= 'Z';
  when "011" =>
    dout := F;
    RAM(Badd) := F;
    Q0 <= 'Z';
    Q3 <= 'Z';
    RAM0 <= 'Z';
    RAM3 <= 'Z';
  when "100" =>
    dout := F;
    RAM(Badd) := RAM3 & F(3 downto 1);
    Q := Q3 & Q(3 downto 1);
    Q0 <= 'Z';
    RAM3 <= 'Z';
    RAM0 <= F(0); -- SHIFTER SIGNALS
    Q0 <= Q(0);
  when "101" =>
    dout := F;
    RAM(Badd) := RAM3 & F(3 downto 1);
    Q0 <= 'Z';
    Q3 <= 'Z';
    RAM3 <= 'Z';
    RAM0 <= F(0);
  when "110" =>
    dout := F;
    RAM(Badd) := F(2 downto 0) & RAM0;
    Q := Q(2 downto 0) & Q0;
    Q0 <= 'Z';
    RAM0 <= 'Z';
    RAM3 <= F(3);
    Q3 <= Q(3);
  when "111" =>
    dout := F;
    RAM(Badd) := F(2 downto 0) & RAM0;
    Q0 <= 'Z';
    Q3 <= 'Z';
    RAM0 <= 'Z';
    RAM3 <= F(3);
  when others =>
    end case;

-- GENERATE DATA OUTPUT *Y* FROM INTERMEDIATE OUTPUT *dout*.

if (OEBar = '0') then
  Y <= dout;
else
  Y <= "XXXX";
end if;

end process;
end a2901;

```



```

-- AMD 2910 Benchmark
-- Source: AMD data book
-- VHDL Benchmark author Indrancel Ghosh
-- University of California, Irvine, CA 92717
-- Developed on Feb 19, 1992
-- Verification Information:
--
-- Syntax      Verified      By whom?      Date      Simulator
-- Functionality  yes      Champaka Ramachandran  Sept17, 92  ZYCAD
--              yes      Champaka Ramachandran  Sept17, 92  ZYCAD)
-----
use work.types.all;
use work.MVI7_functions.all;
use work.synthesis_types.all;

entity AM2910 is
port (
    I : in MVL7_VECTOR(3 downto 0); -- 2910 instruction
    CCEN_BAR : in MVL7; -- condition code enable input bit
    CC_BAR : in MVL7; -- condition code input bit
    RLD_HAR : in MVL7; -- R register load
    CI : in MVL7; -- carry in
    OEBar : in MVL7; -- tri-state driver
    clk : in clock; -- clock
    D : in MVL7_VECTOR(11 downto 0); -- direct inputs
    Y : out MVL7_VECTOR(11 downto 0); -- output instruction word
    PL_BAR : out MVL7; --
    VECT_BAR : out MVL7; --
    MAP_BAR : out MVL7; --
    FULL_HAR : out MVL7; -- stack full flag
);
end AM2910;

architecture AM2910 of AM2910 is
begin
process
    variable FAIL : MVL7; -- CC fail flag
    variable SP : INTEGER range 0 to 5; -- stack pointer
    variable STACK : MEMORY_12_bit(5 downto 0); -- stack register file
    variable RK : MVL7_vector(11 downto 0);
    variable uPC : MVL7_vector(11 downto 0);
    variable Y_temp : MVL7_vector(11 downto 0);
begin
    wait until ( clk = '0') and (not clk'stable);
    fail := CC_bar and ( not CCEN_bar);

    if (SP = 5) then -- NECESSARY FOR CORRECT SIMULATION
        FULL_BAR <= '0'; -- SINCE THIS PROCESS IS NOT TRIGGERED BY
    else
        FULL_HAR <= '1'; -- A RISING CLOCK EDGE
    end if;

```

```

-----
case I is
when '0000' => -- JZ instruction
    Y_temp := '000000000000';
    if (RLD_HAR = '0') then
        RK := D;
    end if;
    SP := 0;
    uPC := '000000000000';
    MAP_BAR <= '1';
    VECT_BAR <= '1';
    PL_BAR <= '0';
-----
when '0001' => -- JNS instruction
    if (FAIL = '0') then
        Y_Lemp := D;
        if (SP /= 5) then -- PUSH
            SP := SP + 1;
        end if;
        STACK(SP) := uPC;
    else
        Y_Lemp := uPC;
    end if;
    if (RLD_HAR = '0') then
        RK := D;
    end if;
    uPC := Y_temp + ('000000000000' & CI);
    MAP_BAR <= '1';
    VECT_BAR <= '1';
    PL_BAR <= '0';
-----
when '0010' => -- JMAP instruction
    Y_temp := D;
    if (RLD_HAR = '0') then
        RK := D;
    end if;
    uPC := Y_temp + ('000000000000' & CI);
    MAP_BAR <= '0';
    VECT_BAR <= '1';
    PL_BAR <= '1';
-----
when '0011' => -- CJZ instruction
    if (FAIL = '1') then
        Y_Lemp := uPC;
    else
        Y_temp := D;
    end if;
    if (RLD_HAR = '0') then

```

```

        RK := D;
    end if;
    uPC := Y_Lemp + ('000000000000' & CI);
    MAP_BAR <= '1';
    VECT_BAR <= '1';
    PL_BAR <= '0';
-----
when '0100' => -- PUSH instruction
    Y_temp := uPC;
    if ( FAIL = '0') or (RLD_HAR = '0') then
        RE := D;
    end if;
    if (SP /= 5) then -- PUSH
        SP := SP + 1;
    end if;
    STACK(SP) := uPC;
    uPC := Y_temp + ('000000000000' & CI);
    MAP_HAR <= '1';
    VECT_HAR <= '1';
    PL_BAR <= '0';
-----
when '0101' => -- JSRP instruction
    if (FAIL = '1') then
        Y_temp := RE;
    else
        Y_Lemp := D;
    end if;
    if (RLD_BAR = '0') then
        RK := D;
    end if;
    if (SP /= 5) then -- PUSH
        SP := SP + 1;
    end if;
    STACK(SP) := uPC;
    uPC := Y_Lemp + ('000000000000' & CI);
    MAP_BAR <= '1';
    VECT_BAR <= '1';
    PL_HAR <= '0';
-----
when '0110' => -- CJV instruction
    if (FAIL = '1') then
        Y_Lemp := uPC;
    else
        Y_temp := D;
    end if;
    if (RLD_HAR = '0') then
        RE := D;
    end if;
    uPC := Y_Lemp + ('000000000000' & CI);

```

```

    MAP_HAR <= '1';
    VECT_HAR <= '0';
    PL_BAR <= '1';
-----
when '0111' => -- JRP instruction
    if (FAIL = '1') then
        Y_temp := RE;
    else
        Y_Lemp := D;
    end if;
    if (RLD_BAR = '0') then
        RK := D;
    end if;
    uPC := Y_temp + ('000000000000' & CI);
    MAP_HAR <= '1';
    VECT_BAR <= '1';
    PL_BAR <= '0';
-----
when '1000' => -- RPOP instruction
    if (RE = '000000000000') then
        Y_temp := uPC;
        if (SP /= 0) then -- POP
            SP := SP - 1;
        end if;
    else
        Y_Lemp := STACK(SP);
        if (RLD_BAR = '1') then
            RK := RK - '000000000001';
        end if;
    end if;
    if (RLD_BAR = '0') then
        RE := D;
    end if;
    uPC := Y_temp + ('000000000000' & CI);
    MAP_HAR <= '1';
    VECT_BAR <= '1';
    PL_BAR <= '0';
-----
when '1001' => -- RPOP instruction
    if (RE /= '000000000000') then
        Y_temp := D;
        if (RLD_HAR = '1') then
            RE := RE - '000000000001';
        end if;
    else
        Y_Lemp := uPC;
    end if;
    if (RLD_HAR = '0') then
        RK := D;
    end if;
    uPC := Y_Lemp + ('000000000000' & CI);

```

```
MAP_BAR <= '1';
VECT_BAR <= '1';
PL_BAR <= '0';
```

```
MAP_BAR <= '1';
VECT_BAR <= '1';
PI_BAR <= '0';
```

```
when '1010' => -- CRTN instruction
```

```
if (FAIL = '0') then
  Y_Lemp := STACK(SP);
  if (SP /= 0) then
    SP := SP - 1;
  end if;
else
  Y_Lemp := uPC;
end if;
if (RIID_BAR = '0') then
  RK := D;
end if;
uPC := Y_Lemp + ('00000000000' & CI);
```

```
MAP_BAR <= '1';
VECT_BAR <= '1';
PI_BAR <= '0';
```

```
when '1011' => -- CJPP instruction
```

```
if (FAIL = '0') then
  Y_Lemp := D;
  if (SP /= 0) then
    SP := SP - 1;
  end if;
else
  Y_Lemp := uPC;
end if;
if (RIID_BAR = '0') then
  RE := D;
end if;
uPC := Y_Lemp + ('00000000000' & CI);
```

```
MAP_BAR <= '1';
VECT_BAR <= '1';
PI_BAR <= '0';
```

```
when '1100' => -- LDCT instruction
```

```
Y_Lemp := uPC;
RK := D;
uPC := Y_Lemp + ('00000000000' & CI);
```

```
MAP_BAR <= '1';
VECT_BAR <= '1';
PL_BAR <= '0';
```

```
when '1101' => -- LOOP instruction
```

```
if (FAIL = '0') then
  Y_Lemp := uPC;
  if (SP /= 0) then
    SP := SP - 1;
  end if;
else
  Y_Lemp := STACK(SP);
end if;
```

```
if (RIID_BAR = '0') then
  RE := D;
end if;
uPC := Y_Lemp + ('00000000000' & CI);
MAP_BAR <= '1';
VECT_BAR <= '1';
PI_BAR <= '0';
```

```
when '1110' => -- CONT instruction
```

```
Y_Lemp := uPC;
if (RIID_BAR = '0') then
  RE := D;
end if;
uPC := Y_Lemp + ('00000000000' & CI);
MAP_BAR <= '1';
VECT_BAR <= '1';
PI_BAR <= '0';
```

```
when '1111' => -- TWB instruction
```

```
if RK = '00000000000' then
  if fail = '1' then
    Y_Lemp := D;
  else
    Y_Lemp := uPC;
  end if;
  if (SP /= 0) then
    SP := SP - 1;
  end if;
else
  if (FAIL = '0') then
    Y_Lemp := uPC;
    if (SP /= 0) then
      SP := SP - 1;
    end if;
  else
    Y_Lemp := stack(sp);
  end if;
  if (RLD_BAR = '1') then
    RK := RK - '00000000001';
  end if;
end if;
if (RIID_BAR = '0') then
  RE := D;
end if;
uPC := Y_Lemp + ('00000000000' & CI);
```

```
when others =>
  end case;
```

```
-- TRI-STATE DRIVER CONTROL
```

```
if OEbar = '0' then
  Y <= Y_Lemp;
else
  Y <= 'ZZZZZZZZZZZZ';
end if;
end process;
end AM2910;
```

```

-----
-- Intel 8251 Benchmark -- Complete design model
-- Source: Intel Data Book
-- VHDL Benchmark author Indraneel Ghosh
-- University Of California, Irvine, CA 92717
-- Developed on April 7, 92
-- Verification Information:
--
-- Verified      By whom?      Date      Simulator
-----
-- Syntax       yes      ChampaKa Ramachandran  Sept 18, 92  ZYCAD
-- Functionality yes      Chumpuka Ramuchandrun  Sept 18, 92  ZYCAD
-----

```

```

use work.types.all;
use work.MV17_functions.all;
use work.synthesis_types.all;

```

```

entity Intel_8251 is
port (
CLK      : in clock;
RXC_BAR  : in clock;
TXC_BAR  : in clock;
RESET    : in MVL7;
CS_HAR   : in MVL7;
C_D_HAR  : in MVL7;
RD_BAR   : in MVL7;
WR_BAR   : in MVL7;
Rx()     : in MVL7;
Tx()     : out MVL7;
D_0      : inout MVL7;
D_1      : inout MVL7;
D_2      : inout MVL7;
D_3      : inout MVL7;
D_4      : inout MVL7;
D_5      : inout MVL7;
D_6      : inout MVL7;
D_7      : inout MVL7;
TxEMPTY  : out MVL7;
TxRDY    : out MVL7;
SYNDET_BD : inout MVL7;
RxRDY    : out MVL7;
DPK_HAR  : out MVL7;
RTS_BAR  : out MVL7;
DSR_BAR  : in MVL7;
CTS_HAR  : in MVL7
);
end;

```

```

architecture USAKT of Intel_8251 is
signal mode          : MV17_VECTOR(7 downto 0);
signal command       : MVL7_VECTOR(7 downto 0);
signal SYNC1         : MVL7_VECTOR(7 downto 0);
signal SYNC2         : MVL7_VECTOR(7 downto 0);
signal SYNC_mask     : MVL7_VECTOR(7 downto 0);
signal Tx_buffer     : MVL7_VECTOR(7 downto 0);
signal Rx_buffer     : MVL7_VECTOR(7 downto 0);
signal Tx_wr_while_cts : MVL7;
signal baud_clocks   : MV17_VECTOR(7 downto 0);
signal stop_clocks   : MVL7_VECTOR(7 downto 0);

```

```

signal brk_clocks    : MV17_VECTOR(10 downto 0);
signal chars         : MV17_VECTOR(3 downto 0);
signal SYNDET_BD_temp : MVL7;
signal status_main   : MVL7_VECTOR(7 downto 0); -- sub-signal (main)
signal status_Rx     : MVL7_VECTOR(7 downto 0); -- sub-signal (Rx)
signal status_Tx     : MVL7_VECTOR(7 downto 0); -- sub-signal (Tx)
signal status        : MVL7_VECTOR(7 downto 0);
signal trigger_status_main : MVL7 := '0'; -- trigger-signal (main)
signal trigger_status_Tx  : MVL7 := '0'; -- trigger-signal (Tx)
signal trigger_status_Rx  : MVL7 := '0'; -- trigger-signal (Rx)
signal SYNDET_HD_Rx      : MVL7; -- sub-signal (Rx)
signal SYNDET_BD_main    : MVL7; -- sub-signal (main)
signal trigger_SYNDET_BD_main : MVL7 := '0'; -- trigger-signal (main)
signal trigger_SYNDET_HD_Rx : MVL7 := '0'; -- trigger-signal (Rx)
signal RxRDY_main        : MVL7; -- sub-signal (Rx)
signal RxRDY            : MVL7; -- sub-signal (main)
signal trigger_RxRDY_main : MVL7 := '0'; -- trigger-signal (main)
signal trigger_RxRDY_Rx  : MVL7 := '0'; -- trigger-signal (Rx)

```

```

begin
-- *****
main : process
-- *****

```

```

variable mode_var          : MV17_VECTOR(7 downto 0);
variable status_var       : MVL7_VECTOR(7 downto 0);
variable command_var      : MVL7_VECTOR(7 downto 0);
variable baud_clocks_var  : MV17_VECTOR(7 downto 0);
variable stop_clocks_var  : MV17_VECTOR(7 downto 0);
variable chars_var        : MVL7_VECTOR(3 downto 0);

```

```

-- Because signals dont get new values immediately on assignment, we need to
-- use variables (mode_var, command_var, baud_clocks_var, stop_clocks_var,
-- status_var, chars_var)
-- which are the same as signals
-- (mode, command, stop_clocks, stop_clocks, status, chars).

```

```

-- This is needed because the new values of these signals are used for
-- further computation inside the "main" process.

```

```

variable next_cpu_control_word : MV17_VECTOR(1 downto 0);
-- Variable "next_cpu_control_word" keeps track of which control
-- word should come next from the CPU (mode/SYNC-char/command)
-- 00 = mode
-- 01 = SYNC CHAR 1
-- 10 = SYNC CHAR 2
-- 11 = command

```

```

variable SYNC_var : MV17_VECTOR(7 downto 0);
variable temp      : MV17_VECTOR(10 downto 0);

```

```

begin
wait until ( clk = '1' ) and ( not clk'stable );
if (CS_BAR = '0') then -- if chip select
if ( RKS_HAR = '1' ) or ( command_var(6) = '1' ) then -- if reset (external)
-- Initialize ports and global
-- signals on reset
DPK_HAR <= '1';

```

```

RTS_HAR <= '1';
command_var := '00000000';
command <= command_var;
status_var := '00000101';
status_main <= status_var;
trigger_status_main <= not(trigger_status_main);
Tx_wr_while_cts <= '0';
-- Note the type of control word that comes next
-- (Mode word)

```

```

next_cpu_control_word := '00';
else -- if not reset
if (RD_BAR = '0') then -- if read
if (C_D_BAR = '1') then -- if read status
-- read the value at the DSR_HAR input

```

```

status_var := not(DSR_BAR) & status(6 downto 0);
-- Place status word on data bus pins
D_0 <= status_var(0);
D_1 <= status_var(1);
D_2 <= status_var(2);
D_3 <= status_var(3);
D_4 <= status_var(4);
D_5 <= status_var(5);
D_6 <= status_var(6);
D_7 <= status_var(7);

```

```

if ( mode_var(1 downto 0) = '00' ) then -- Sync mode
SYNDET_BD_main <= '0'; -- reset SYNDET_BD on status read
trigger_SYNDET_HD_main <= not(trigger_SYNDET_HD_main);
status_var := status(7) & '0' & status(5 downto 0);
status_main <= status_var;
trigger_status_main <= not(trigger_status_main);
end if;

```

```

else -- if read Rx data
if (command_var(2) = '1') then -- if RxENABLE

```

```

-- Place received data character on data bus pins
D_0 <= Rx_buffer(0);
D_1 <= Rx_buffer(1);
D_2 <= Rx_buffer(2);
D_3 <= Rx_buffer(3);
D_4 <= Rx_buffer(4);
D_5 <= Rx_buffer(5);
D_6 <= Rx_buffer(6);
D_7 <= Rx_buffer(7);

```

```

RxRDY_main <= '0'; -- Reset RxRDY on data read
trigger_RxRDY_main <= not(trigger_RxRDY_main);
status_var := status(7 downto 2) & '0' & status(0);
status_main <= status_var;
trigger_status_main <= not(trigger_status_main);
end if;

```

```

end if; -- end if command/data

```

```

elsif (WR_HAR = '0') then -- if write
-- Tristate the data bus pins (bi-directional)
-- so that CPU can write data/control word
D_0 <= 'Z';
D_1 <= 'Z';
D_2 <= 'Z';
D_3 <= 'Z';
D_4 <= 'Z';
D_5 <= 'Z';
D_6 <= 'Z';
D_7 <= 'Z';

```

```

wait for 0 ns; -- only for simulation (resolution function)

```

```

if (C_D_BAR = '1') then -- if write command/mode/sync-char
case (next_cpu_control_word) is

```

```

when '00' => -- next_cpu_control_word = mode
-- Read mode word from data bus lines
mode_var(0) := D_0;
mode_var(1) := D_1;
mode_var(2) := D_2;
mode_var(3) := D_3;
mode_var(4) := D_4;
mode_var(5) := D_5;
mode_var(6) := D_6;
mode_var(7) := D_7;
mode <= mode_var;

```

```

-- Find the number of bits per character
chars_var := '0101' + ('00' & mode_var(3 downto 2) );
chars <= chars_var; -- no. of char bits

```

```

if ( mode_var(1 downto 0) = '00' ) then -- Sync mode
-- Note the type of control word that comes next
if ( mode_var(6) = '1' ) then -- Ext Sync Mode
next_cpu_control_word := '11'; -- command word
else -- Int Sync Mode
next_cpu_control_word := '01'; -- SYNC1
end if;

```

```

stop_clocks <= '00000000';
stop_clocks_var := '00000000';
baud_clocks <= '00000001';
baud_clocks_var := '00000001';

```

```

else -- if Async mode
-- Note the type of control word that comes next
next_cpu_control_word := '11'; -- command
-- Find the number of clock cycles per data/parity b
case ( mode_var(1 downto 0) ) is -- set baud rate clks

```

```

when '00' =>
when '01' =>
    baud_clocks_var := '00000001';
    baud_clocks <= baud_clocks_var;
when '10' =>
    baud_clocks_var := '00010000';
    baud_clocks <= baud_clocks_var;
when '11' =>
    baud_clocks_var := '01000000';
    baud_clocks <= baud_clocks_var;
when others =>
    baud_clocks <= baud_clocks_var;
end case;
-- Find the number of stop bit clock cycles
case ( mode_var(7 downto 6) ) is -- set stop bit clks
when '00' =>
when '01' =>
    stop_clocks_var := baud_clocks_var;
    stop_clocks <= stop_clocks_var;
when '10' =>
    stop_clocks_var := baud_clocks_var(7 downto 0) +
        ( '0' & baud_clocks_var(7 downto 0) );
    stop_clocks <= stop_clocks_var;
when '11' =>
    stop_clocks_var := baud_clocks_var(6 downto 0) &
        stop_clocks <= stop_clocks_var;
when others =>
end case;
-- Calculate no. of clocks that Hx1) has to be low for a break to be detected.
-- (Two full character sequences)
-- Count number of start bit clocks
temp := '000' & baud_clocks_var;
-- Count number of data bit clocks (full character)
while ( chars_var /= '0000' ) loop
    temp := temp + ( '000' & baud_clocks_var );
    chars_var := chars_var - '0001';
end loop;
-- Count number of parity bit clocks
if (mode_var(4) = '1') then -- if Parity enable
    temp := temp + ( '000' & baud_clocks_var );
end if;
-- Count number of stop bit clocks
temp := temp + ( '000' & stop_clocks_var );
-- Double this number (Hx1) has to be low through two
-- character sequences)
brk_clocks <= temp(9 downto 0) & '0';
end if; -- end if sync mode
when '01' => -- next_cpu_control_word = SYNC-CHAR 1
-- Read the SYNC1 character from the data bus lines
SYNC_var(0) := D_0;
SYNC_var(1) := D_1;
SYNC_var(2) := D_2;
SYNC_var(3) := D_3;
SYNC_var(4) := D_4;
SYNC_var(5) := D_5;
SYNC_var(6) := D_6;
SYNC_var(7) := D_7;
-- Note the type of control word that comes next
if (mode_var(7) = '0') then -- if Double SYNC char
    next_cpu_control_word := '10'; -- SYNC2
else
    next_cpu_control_word := '11'; -- Command
end if;
-- Place SYNC1 character into proper format
-- (according to number of bits per character).
-- Also create a template (SYNC_mask) to be used in SYNC-character
case (mode_var(3 downto 2)) is -- char. length
when '00' =>
    SYNC1 <= '000' & SYNC_var(4 downto 0);
    SYNC_mask <= '00011111';
when '01' =>
    SYNC1 <= '00' & SYNC_var(5 downto 0);
    SYNC_mask <= '00111111';
when '10' =>
    SYNC1 <= '0' & SYNC_var(6 downto 0);
    SYNC_mask <= '01111111';
when '11' =>
    SYNC1 <= SYNC_var;
    SYNC_mask <= '11111111';
when others =>
end case;
when '10' => -- next_cpu_control_word = SYNC-CHAR 2
-- Read the SYNC2 character from the data bus lines
SYNC_var(0) := D_0;
SYNC_var(1) := D_1;
SYNC_var(2) := D_2;
SYNC_var(3) := D_3;
SYNC_var(4) := D_4;
SYNC_var(5) := D_5;
SYNC_var(6) := D_6;
SYNC_var(7) := D_7;
-- Note the type of control word that comes next (command)
next_cpu_control_word := '11';
-- Place SYNC2 character into proper format
-- (according to number of bits per character).
case (mode_var(3 downto 2)) is -- char. length
when '00' =>
    SYNC2 <= '000' & SYNC_var(4 downto 0);
when '01' =>
    SYNC2 <= '00' & SYNC_var(5 downto 0);
when '10' =>
    SYNC2 <= '0' & SYNC_var(6 downto 0);
when '11' =>
    SYNC2 <= SYNC_var;
when others =>
end case;
when '11' => -- next_cpu_control_word = command
-- Read the command word from the data bus lines
command_var(0) := D_0;
command_var(1) := D_1;
command_var(2) := D_2;
command_var(3) := D_3;
command_var(4) := D_4;
command_var(5) := D_5;
command_var(6) := D_6;
command_var(7) := D_7;
command <= command_var;
-- Note the type of control word that comes next
-- (another command if there is no reset)
next_cpu_control_word := '11';
status_var := status;
-- If receiver is disabled, reset RxDY
if (command_var(2) = '0') then -- RxENABLE
    RxDY_main <= '0';
    trigger_RxDY_main <= not(trigger_RxDY_main);
    status_var := status(7 downto 2) & '0' & status(0);
end if;
-- Reset error flags (depending on command word)
if (command_var(4) = '1') then -- error reset
    status_var := status_var(7 downto 6) & '000' & status_var(2)
end if;
-- Update status
status_main <= status_var;
trigger_status_main <= not(trigger_status_main);
-- Assert output pins (depending on command word)
RTS_BAR <= not(command_var(5));
DTR_BAR <= not(command_var(1));
when others =>
end case;
else -- if write data for Transmission
if (command_var(0) = '1') then -- if TxENABLE
-- Load data for transmission from data bus lines into parallel b
case (mode_var(3 downto 2)) is -- char. length
when '00' =>
    TX_buffer <= '000' & D_4 & D_3 & D_2 & D_1 & D_0;
when '01' =>
    TX_buffer <= '00' & D_5 & D_4 & D_3 & D_2 & D_1 & D_0;
when '10' =>
    TX_buffer <= '0' & D_6 & D_5 & D_4 & D_3 & D_2 & D_1 & D_0;
when '11' =>
    TX_buffer <= D_7 & D_6 & D_5 & D_4 & D_3 & D_2 & D_1 & D_0;
when others =>
end case;
-- Reset TxRDY status bit after loading data for transmission
status_var := status(7 downto 1) & '0'; -- TxRDY
status_main <= status_var;
trigger_status_main <= not(trigger_status_main);
-- Note whether data was written by CPU while CTS_BAR was low
if (CTS_BAR = '0') then -- Tx data was written while
    Tx_wr_while_cts <= '1'; -- CTS_BAR was asserted
else
    Tx_wr_while_cts <= '0';
end if;
end if; -- end if command/data
else -- if neither read nor write
end if; -- end if read/write
end if; -- end if reset
end if; -- end if chip select
end process main;
-- *****
Transmitter : process
-- *****
variable parity : MVI,7;
variable serial_Tx_buffer : MVL,VECTOR(7 downto 0);
variable store_Tx_buffer : MVL,VECTOR(7 downto 0); -- parity computation
variable clk_count : MVI,VECTOR(7 downto 0);
variable char_bit_count : MVI,VECTOR(3 downto 0);
begin
if ( RESET = '1' ) or ( command(6) = '1' ) then -- if reset
    TxD <= '1'; -- Send marking signal
    TxEMPTY <= '1';
    status_Tx <= status(7 downto 3) & '1' & status(1 downto 0);
    trigger_status_Tx <= not(trigger_status_Tx);
wait until ( TxC_BAR = '0' ) and ( not TxC_BAR'stable );
else
if (status(0) = '0') then -- if Tx_buffer is full
-- (TxRDY status bit reset)
-- If Tx is enabled and CTS_BAR is low or data was written while CTS_BAR was 1
if ( ( CTS_BAR = '0' ) and (command(0) = '1' ) ) or ( Tx_wr_while_cts = '1' )
-- load data into serial buffer
serial_Tx_buffer := Tx_buffer;
store_Tx_buffer := Tx_buffer; -- used for parity computation
-- Reset TxEMPTY and set TxRDY status bit (we are going to start transm
TxEMPTY <= '0';

```

```

if (command(2) = '1') then
  status_TX <= status(7 downto 3) & '0' & status(1) & '1';
else
  status_TX <= status(7 downto 3) & '001';
end if;

trigger_status_TX <= not(trigger_status_TX);
-- TXRDY and TXEMPTY status bits

if (mode(1 downto 0) /= '00') then -- if async mode (start)
  -- SEND START BIT

  clk_count := baud_clocks;

-- Loop for counting number of clock cycles per bit (according to baud)
  while ( clk_count /= '00000000') loop
    TXD <= '0';
    wait until (TxC_BAR = '0') and (not TxC_BAR'stable);
    clk_count := clk_count - '00000001';
  end loop;

end if; -- end if async mode (start)

-- SEND CHARACTER BITS
char_bit_count := chars;

-- Loop for counting number of character bits
while ( char_bit_count /= '0000') loop

  char_bit_count := char_bit_count - '0001';
  clk_count := baud_clocks;

-- Loop for counting number of clock cycles per bit (according to baud)
  while ( clk_count /= '00000000') loop
    TXD <= serial_Tx_buffer(0);
    wait until (TxC_HAR = '0') and (not TxC_HAR'stable);
    clk_count := clk_count - '00000001';
  end loop;

  serial_Tx_buffer := '0' & serial_Tx_buffer(7 downto 1);

end loop;

-- SEND PARITY BIT (IF APPLICABLE)
if (mode(4) = '1') then -- if parity enabled

-- CALCULATE PARITY BIT
  parity := store_Tx_buffer(0) xor store_Tx_buffer(1) xor
            store_Tx_buffer(2) xor store_Tx_buffer(3) xor
            store_Tx_buffer(4) xor store_Tx_buffer(5) xor
            store_Tx_buffer(6) xor store_Tx_buffer(7) xor
            (not mode(5));

  clk_count := baud_clocks; -- SEND PARITY BIT

-- Loop for counting number of clock cycles per bit (according to baud)
  while ( clk_count /= '00000000') loop
    TXD <= parity;
    wait until (TxC_HAR = '0') and (not TxC_HAR'stable);
    clk_count := clk_count - '00000001';
  end loop;

end if; -- end if parity enabled

-- Data was sent. Set TXEMPTY unless a new data char has been written and is
if ( not(((CTS_HAR = '0') and (command(0) = '1')) or (Tx_wr_while_cts
and (status(0) = '0')))) then

  TXEMPTY <= '1';
  status_TX <= status(7 downto 3) & '1' & status(1 downto 0);
  trigger_status_TX <= not(trigger_status_TX);

end if;

if (mode(1 downto 0) /= '00') then -- if async mode (stop)

-- SEND STOP BIT
  clk_count := stop_clocks;

-- Loop for counting number of clock cycles in stop stop bit
  while ( clk_count /= '00000000') loop
    TXD <= '1';
    wait until (TxC_HAR = '0') and (not TxC_HAR'stable);
    clk_count := clk_count - '00000001';
  end loop;

end if; -- end if async mode (stop)

else -- if Transmitter not enabled or data was written while CTS_BAR w
  TXD <= '1'; -- mark
  TXEMPTY <= '1';

  wait until ( TxC_HAR = '0' ) and ( not TxC_HAR'stable );

end if; -- end if Tx disable and data was written while it was disabled

else -- if Tx_buffer empty
  TXEMPTY <= '1';

  if (command(3) = '1') then -- if send break
    TXD <= '0';
    wait until ( TxC_BAR = '0' ) and ( not TxC_BAR'stable );
  else -- if dont send break
    if (mode(1 downto 0) = '00') then -- if Sync mode
      if (CTS_BAR = '0') and (command(0) = '1') then -- if Tx enabled

-- SEND SYNC1
        serial_Tx_buffer := SYNC1;
        store_Tx_buffer := SYNC1; -- for parity
        char_bit_count := chars;

-- SEND CHARACTER BITS
-- Loop for counting number of character bits

        while ( char_bit_count /= '0000') loop
          char_bit_count := char_bit_count - '0001';
          clk_count := baud_clocks;
        end loop;
      end if;
    end if;
  end if;
end if;

-- Loop for counting number of clock cycles per bit (according to baud)
  while ( clk_count /= '00000000') loop
    TXD <= serial_Tx_buffer(0);
    wait until (TxC_BAR = '0') and (not TxC_BAR'stable);
    clk_count := clk_count - '00000001';
  end loop;

  serial_Tx_buffer := '0' & serial_Tx_buffer(7 downto 1);

end loop;

if (mode(4) = '1') then -- if parity enabled

-- CALCULATE PARITY BIT
  parity := store_Tx_buffer(0) xor store_Tx_buffer(1) xor
            store_Tx_buffer(2) xor store_Tx_buffer(3) xor
            store_Tx_buffer(4) xor store_Tx_buffer(5) xor
            store_Tx_buffer(6) xor store_Tx_buffer(7) xor
            (not mode(5)); -- even/odd parity

  clk_count := baud_clocks;

-- SEND PARITY BIT

-- Loop for counting number of clock cycles per bit (according
  while ( clk_count /= '00000000') loop
    TXD <= parity;
    wait until (TxC_BAR = '0') and (not TxC_BAR'stable);
    clk_count := clk_count - '00000001';
  end loop;

end if; -- end if parity enabled

end if; -- end if Double Sync

else -- if Tx disabled
  TXD <= '1'; -- Send marking signal
  wait until ( TxC_HAR = '0' ) and ( not TxC_HAR'stable );

end if;

else -- if Async mode
  TXD <= '1'; -- Send marking signal
  wait until ( TxC_HAR = '0' ) and ( not TxC_HAR'stable );

end if; -- end if Sync mode

end if; -- end if send break

end if; -- end if Tx_buffer full

end if; -- end if reset

end process transmitter;

-- .....
receiver : process
-- .....

variable serial_Rx_buffer : MVI7_VECTOR(7 downto 0);
variable sync_shift : MVI7_VECTOR(7 downto 0);
variable brk_count : MVI7_VECTOR(10 downto 0);
variable clk_count : MVI7_VECTOR(7 downto 0);
variable half_baud : MVI7_VECTOR(7 downto 0);
variable char_bit_count : MVI7_VECTOR(3 downto 0);
variable status_var : MVI7_VECTOR(7 downto 0);
variable got_sync : MVI7; -- This variable is used in enter hunt
-- mode to check whether
-- synchronization has been achieved i
-- (Used in Internal Sync detect Mode)
variable got_half_sync : MVI7; -- This variable is used in Double
-- Sync mode (outside hunt mode). Its
-- assertion means that SYNC1 has been
-- received and SYNDET_BD should be
-- asserted if SYNC2 is received next

variable parity : MVI7;

```



```

    if (command(0) = '1') then
      status_var := status_var(7) & '1' & status_var(5 downto 0);
    else
      status_var := status_var(7) & '1' & status_var(5 downto 3) &
        '1' & status_var(1 downto 0);
    end if;

    end if; -- end if double sync mode
  end if; -- end if we got SYNC1

  -- transfer received character to parallel buffer
  RX_buffer <= serial_RX_buffer;

  -- Check if RXRDY was already set (i.e. previous character unread)
  if (status(1) = '1') then

    -- Set Overrun Error flag if previous character was unread
    if (command(0) = '1') then
      status_var := status_var(7 downto 5) & '1' & status_var(3 downto
        0);
    else
      status_var := status_var(7 downto 5) & '1' & status_var(3) &
        '1' & status_var(1 downto 0);
    end if;

  else
    -- Set RXRDY to tell CPU to read new character
    RXRDY_Rx <= '1';
    trigger_HxRDY_Rx <= not(trigger_HxRDY_Rx);
    if (command(0) = '1') then
      status_var := status_var(7 downto 2) & '1' & status_var(0);
    else
      status_var := status_var(7 downto 3) & '11' & status_var(0);
    end if;

  end if; -- end if Rx buffer full

  status_Rx <= status_var;
  trigger_status_Rx <= not(trigger_status_Rx);
else
  -- ASYNCHRONOUS MODE
  -- Check whether RXD is high. If so, then it is ready to
  -- receive the Start Bit (low) of the next character
  if (RXD = '1') then

    -- Set Break Detect (SYNDET_BD) low if RXD is high
    brk_count := '00000000000';
    SYNDET_BD_Rx <= '0';
    trigger_SYNDET_BD_Rx <= not(trigger_SYNDET_BD_Rx);

    if (command(0) = '1') then
      status_Rx <= status(7) & '0' & status(5 downto 0);
    else
      status_Rx <= status(7) & '0' & status(5 downto 3) & '1' &
        status(1 downto 0);
    end if;
  end if;

```

```

    trigger_status_Rx <= not(trigger_status_Rx);
  -- WAIT FOR FALLING EDGE ON RXD (START BIT) IN CASE A RESET (INT/EXT) OCCURS
  wait until ((RXD = '0') and (not RXD'stable)) or (RESET = '1') or (co
  -- if not reset
  if ((RKSMT = '0') and (command(6) = '0')) then

    -- START BIT
    -- To sample Start Bit at its mid-point (16X or 64X baud rate
    -- only), wait for half the number of clock cycles per bit
    -- (equal to variable 'half_baud')
    -- Note: Variable 'half_baud' is 0 for 1X baud rate, so we
    -- introduce a separate wait for the 1X mode. (***)
    half_baud := '0' & baud_clocks(7 downto 1);
    clk_count := half_baud;

    -- Loop to wait for half the number of clock cycles per bit
    while (clk_count /= '00000000') loop
      wait until (RxC_HAR = '1') and (not RxC_HAR'stable);
      clk_count := clk_count - '00000001';
    end loop;

    -- Sample Start Bit at its mid-point (Pulse Start Bit Detection
    -- If its a real Start Bit
    if (RXD = '0') then

      -- For 1X baud rate, we introduce a separate wait (as mentioned
      if (mode(1 downto 0) = '01') then
        wait until (RxC_BAR = '1') and (not RxC_BAR'stable);
      end if;

      -- Loop to wait for half the number of clock cycles per bit
      clk_count := half_baud; -- half_baud is 0 for 1X mode

      while (clk_count /= '00000000') loop
        wait until (RxC_HAR = '1') and (not RxC_HAR'stable);
        clk_count := clk_count - '00000001';
      end loop; -- END OF START BIT

      brk_count := brk_count + ('000' & baud_clocks);

    -- ASSEMBLE CHARACTER BITS
    serial_RX_buffer := '000000000';
    char_bit_count := chars;

    -- Loop for counting number of character bits
    while (char_bit_count /= '0000') loop

      -- To sample a Character Bit at its mid-point (16X or 64X baud
      -- rate only), wait for half the number of clock cycles per bit
      -- (equal to variable 'half_baud')
      -- Note: Variable 'half_baud' is 0 for 1X baud rate, so we
      -- introduce a separate wait for the 1X mode. (***)
      clk_count := half_baud;

```

```

    -- Loop to wait for half the number of clock cycles per bit
    while (clk_count /= '00000000') loop
      wait until (RxC_HAR = '1') and (not RxC_HAR'stable);
      clk_count := clk_count - '00000001';
    end loop;

    -- For 1X baud rate, we introduce a separate wait (as me
    if (mode(1 downto 0) = '01') then
      wait until (RxC_BAR = '1') and (not RxC_BAR'stable);
    end if;

    -- Sample character bit at its nominal center
    serial_RX_buffer := RXD & serial_RX_buffer(7 downto 1);

    if (RXD = '1') then

      -- Set Break Detect (SYNDET_BD) low if RXD is high
      brk_count := '00000000000';
      SYNDET_BD_Rx <= '0';
      trigger_SYNDET_BD_Rx <= not(trigger_SYNDET_BD_Rx);

      if (command(0) = '1') then
        status_var := status(7) & '0' & status(5 downto 0);
      else
        status_var := status(7) & '0' & status(5 downto 3) & '1' &
          status(1 downto 0);
      end if;

      status_Rx <= status_var;
      trigger_status_Rx <= not(trigger_status_Rx);
    else
      -- If RXD is low, increase 'brk_count' by the number of clock cy
      brk_count := brk_count + ('000' & baud_clocks);
    end if;

    clk_count := half_baud; -- NOTE: half_baud = 0 for 1X baud

    -- Loop to wait for half the number of clock cycles per bit
    while (clk_count /= '00000000') loop
      wait until (RxC_BAR = '1') and (not RxC_BAR'stable);
      clk_count := clk_count - '00000001';
    end loop;

    char_bit_count := char_bit_count - '0001';

  end loop;

  -- ALIGN ASSEMBLED CHARACTER CORRECTLY
  case (mode(3 downto 2)) is -- char. length
    when '00' =>
      serial_RX_buffer := '000' & serial_RX_buffer(7 downto 3);
    when '01' =>
      serial_RX_buffer := '00' & serial_RX_buffer(7 downto 2);
    when '10' =>
      serial_RX_buffer := '0' & serial_RX_buffer(7 downto 1);
    when '11' =>
      serial_RX_buffer := serial_RX_buffer(7 downto 0);
    when others =>
      serial_RX_buffer := serial_RX_buffer(7 downto 0);
  end case;

```

```

  -- PARITY BIT
  if (mode(4) = '1') then -- if parity enabled

    -- To sample a Parity Bit at its mid-point (16X or 64X baud
    -- rate only), wait for half the number of clock cycles per bit
    -- (equal to variable 'half_baud') Note: Variable 'half_baud' is
    -- 0 for 1X baud rate, so we introduce a separate wait for the 1X m
    clk_count := half_baud;

    -- Loop to wait for half the number of clock cycles per bit
    while (clk_count /= '00000000') loop
      wait until (RxC_HAR = '1') and (not RxC_HAR'stable);
      clk_count := clk_count - '00000001';
    end loop;

    -- For 1X baud rate, we introduce a separate wait (as mention
    if (mode(1 downto 0) = '01') then
      wait until (RxC_BAR = '1') and (not RxC_BAR'stable);
    end if;

    -- CHECK PARITY AT CENTRE OF PARITY BIT
    parity := RXD;

    if (RXD = '1') then

      -- Set Break Detect (SYNDET_BD) low if RXD is high
      brk_count := '00000000000';
      SYNDET_BD_Rx <= '0';
      trigger_SYNDET_BD_Rx <= not(trigger_SYNDET_BD_Rx);

      if (command(0) = '1') then
        status_var := status(7) & '0' & status(5 downto 0);
      else
        status_var := status(7) & '0' & status(5 downto 3) & '1' &
          status(1 downto 0);
      end if;

    else
      -- If RXD is low, increase 'brk_count' by the number of clock cy
      brk_count := brk_count + ('000' & baud_clocks);
    end if;

    -- Verify Parity
    parity := serial_RX_buffer(0) xor serial_RX_buffer(1) xor
      serial_RX_buffer(2) xor serial_RX_buffer(3) xor
      serial_RX_buffer(4) xor serial_RX_buffer(5) xor
      serial_RX_buffer(6) xor serial_RX_buffer(7) xor
      (not mode(5)) xor parity; -- PARITY ERROR

    -- Set Parity Error flag if error is detected
    if (command(0) = '1') then
      status_var := status_var(7 downto 4) & parity & status_var(2 do
    else
      status_var := status_var(7 downto 4) & parity & '1' & status_v
    end if;

    if (mode(1) = '1') then -- if 16X or 64X baud
      status_Rx <= status_var;

```

```

    trigger_status_Rx <= not(trigger_status_Rx);
end if;
-- end if RxENABLE

clk_count := half_baud;      -- half_baud = 0 for 1X baud
-- Loop to wait for half the number of clock cycles per bit
while (clk_count /= '00000000') loop
    wait until (RxC_IAR = '1') and (not RxC_IAR'stable);
    clk_count := clk_count - '00000001';
end loop;

end if;      -- end if parity enabled
-- Transfer received data to parallel buffer
Rx_buffer <= serial_Rx_buffer;
-- Check if RxD was already set (i.e. previous character
-- unread by CPU)
if (status(1) = '1') then      -- if Rx buffer full
-- Set Overrun Error flag if previous character was unread
    if (command(0) = '1') then
        status_var := status_var(7 downto 5) & '1' & status_var(3 downto 0);
    else
        status_var := status_var(7 downto 5) & '1' & status_var(3) & '1' & status_var(1 downto 0);
    end if;
else
-- Set RxDY to tell CPU to read new character
RxRDY_Rx <= '1';
trigger_RxRDY_Rx <= not(trigger_RxRDY_Rx);
    if (command(0) = '1') then
        status_var := status_var(7 downto 2) & '1' & status_var(0);
    else
        status_var := status_var(7 downto 3) & '11' & status_var(0);
    end if;
end if;      -- end if already RxDY
status_Rx <= status_var;
trigger_status_Rx <= not(trigger_status_Rx);
-- STOP BIT(S)
wait until (RxC_BAR = '1') and (not RxC_BAR'stable);
-- check for framing error and break
if (RxD = '1') then
-- Set Break Detect (SYNDET_BD) low if RxD is high
    brk_count := '000000000000';
    SYNDET_BD_Rx <= '0';
    trigger_SYNDET_BD_Rx <= not(trigger_SYNDET_BD_Rx);
    if (command(0) = '1') then
        status_Rx <= status(7) & '0' & status(5 downto 0);
    else
        status_Rx <= status(7) & '0' & status(5 downto 3) & '1' & status(1);
    end if;
    trigger_status_Rx <= not(trigger_status_Rx);
else
-- If RxD is low, set framing error flag.
    if (command(0) = '1') then
        status_Rx <= status(7 downto 6) & '1' & status(4 downto 0);
    else
        status_Rx <= status(7 downto 5) & '1' & status(4 downto 3) & '1' & status(1 downto 0);
    end if;
    trigger_status_Rx <= not(trigger_status_Rx);
-- Increase *brk_count* by the number of clock cycles per bit.
    brk_count := brk_count + ('000' & stop_clocks);
end if;
end if;      -- end if its an actual start bit
end if;      -- end if not reset
else      -- if not yet ready to receive start bit
-- (i.e. RxD is low)
wait until (RxC_IAR = '1') and (not RxC_IAR'stable);
if (RxD = '0') then      -- if still not ready to receive start bit
-- RxD has been low for one more clock cycle. So increment *brk_
    brk_count := brk_count + '000000000001';
-- If RxD has stayed low for two consecutive character sequence lengths,
-- set Break Detect (SYNDET_BD)
    if (brk_count >= brk_clocks) then
        SYNDET_BD_Rx <= '1';
        trigger_SYNDET_BD_Rx <= not(trigger_SYNDET_BD_Rx);
        if (command(0) = '1') then
            status_Rx <= status(7) & '1' & status(5 downto 0);
        else
            status_Rx <= status(7) & '1' & status(5 downto 3) & '1' & status(1);
        end if;
        trigger_status_Rx <= not(trigger_status_Rx);
    end if;
end if;      -- end if break detected
end if;      -- end if still not ready to receive start bit
end if;      -- end if ready to receive start bit
end if;      -- end if sync mode
else      -- if Rx disabled
-- Reset RxDY if receiver is disabled
RxRDY_Rx <= '0';
trigger_RxRDY_Rx <= not(trigger_RxRDY_Rx);
wait until (RxC_IAR = '1') and (not RxC_IAR'stable);
end if;
-- end if RxENABLE
end if;      -- end if reset
end process receiver;
-----
triggering : block
-----
begin
-- The signal *status* and the ports *SYNDET_BD, RxDY* are written
-- to by more than one process. So, we split them up into many
-- *sub-signals* (one for each writing-process).
-- Whenever any process writes to its own *sub-signal*, we assign the
-- new value to the actual signal. This *writing* is monitored by the tr
-- Whenever the signal has to be read, we read the actual signal and not
status <= status_main when (not trigger_status_main'stable) else
    status_Rx when (not trigger_status_Rx'stable) else
    status_Tx when (not trigger_status_Tx'stable) else
    status;
SYNDET_BD_temp <= SYNDET_BD_main when (not trigger_SYNDET_BD_main'stable) else
    SYNDET_BD_Rx when (not trigger_SYNDET_BD_Rx'stable) else
    SYNDET_BD_Temp;
SYNDET_BD <= SYNDET_BD_temp;
RxRDY <= RxRDY_main when (not trigger_RxRDY_main'stable) else
    RxRDY_Rx when (not trigger_RxRDY_Rx'stable) else
    status(1);      -- RxRDY
end block triggering;
-----
TxRDY_pin : block
-----
begin
-- TxRDY pin is dependent on CTS_BAR and TxENABLE, in addition to the Tx
-- Since CTS_IAR can change at any time, we use a separate block for thi
TxRDY <= (not CTS_BAR) and command(0) and status(0);
end block TxRDY_pin;
-----
end USART;

```