

# SYSTEM-LEVEL MODELING OF EQUALIZERS USING VHDL

Rafael de Fermín, Diego Pérez, Fernando Reguero

Alcatel SESA  
RSD R&D Digital Circuits  
Einstein sn, PTM. 28760 Tres Cantos, Madrid. Spain.

## Abstract

*Although the capabilities of VHDL for the definition, modeling and simulation of complex systems are widely admitted, some specific system-level modeling strategies have to be adopted to take advantage of all the power the language can offer. A project where VHDL was used to model a real-life example at behavioral level is presented in this contribution, making a special effort to describe the VHDL style required for such level of abstraction, and focusing on the problems which arose during the encoding of the system in VHDL and on the solutions that were given to these problems.*

## Description of the project: Evaluation of equalizer architectures for a radio link

Very schematically, a signal that is to be transmitted through a radio channel is processed before being sent into the air so that, after crossing the channel, the information can be recovered from the waveform received, which is a transformed version of the original, corrupted with a certain amount of noise. To deal with intersymbol interference and some other effects of the channel on the signal, specially when these effects are non-stationary, an equalizer is added to the receiver. Transmission systems and equalization theory are beyond the scope of this paper, but [1] is an excellent reference about these subjects.

Figure 1 shows the base-band block diagram of a radio link: the signal is modulated and filtered before crossing the channel, and the information is recovered and demodulated at a receiver which includes the equalizer to be

studied. Some distortion and noise-generation blocks were as well considered.

The objective of our project will be to study some equalization architectures for a given radio link, deciding which one best fits our requirements. As the receiver containing the equalizer is the last block in the radio link (see fig. 1), it should now be obvious that the whole system will have to be modelled and simulated, in order to feed the equalizer with a realistic collection of data.

## Tools required

Making a decision about the quality of an equalizer implies the use of some more tools (fig. 2) than just a VHDL simulator, which might be enough for most applications, because a lot of graphical information, like eye patterns or link signatures, will be required. This special feature of the system here presented moved us to use a graphical tool which could represent the results produced by the simulator. For this design, we used a general-purpose tool (fig. 3), which could read data files with a certain structure. The solution proved to be helpful and easy-to-use, once some interfacing routines were developed to translate the data format of the VHDL simulator into the one accepted by our graphical tool.

Finally, a small set of mathematical routines (to calculate Discrete Fourier Transformations, for instance) were also necessary for the analysis of the results. These functions could have been encoded in VHDL and included among the blocks to be simulated; we, however, preferred to use a commercial software to make this kind of

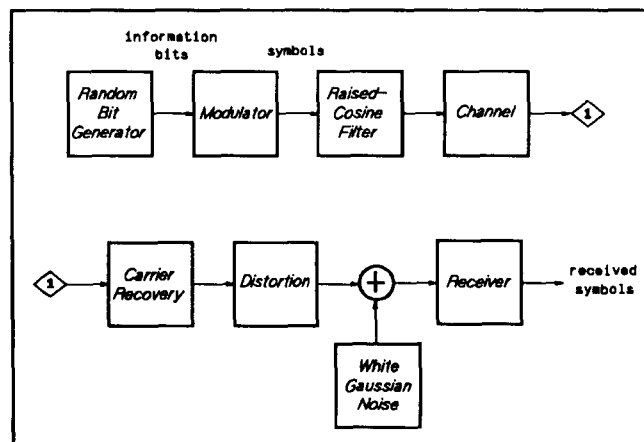


Fig. 1. Block diagram of the radio link.

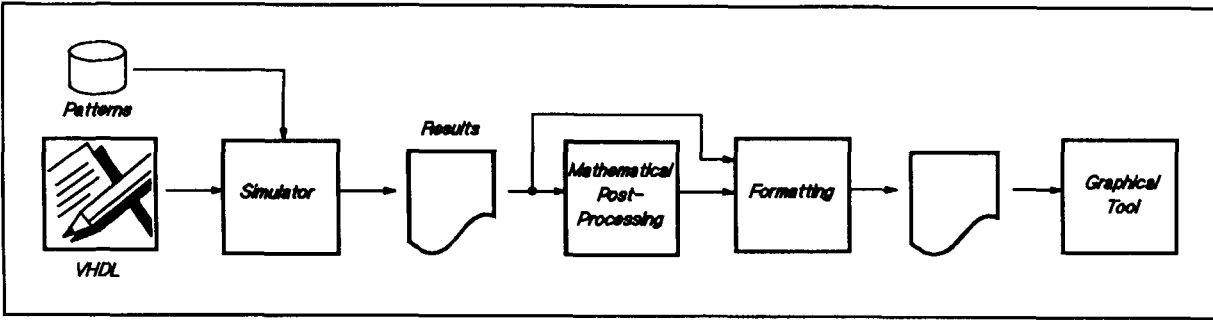


Fig. 2. Tools used for the analysis of the radio link.

calculations, which were handled as a part of the interface between the results of the simulation and the graphical representation of the data.

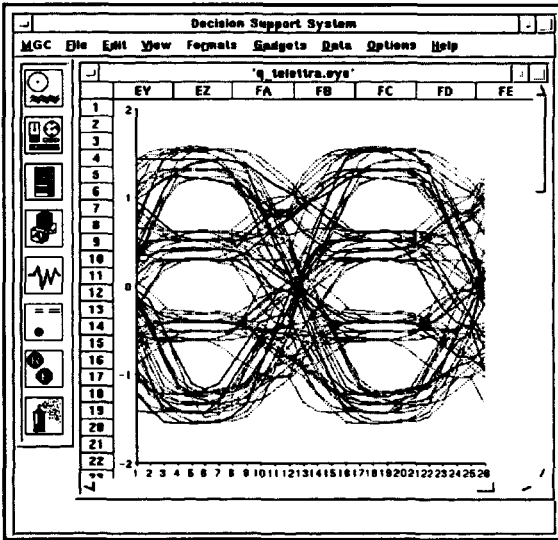


Fig. 3. An example of the information displayed by the graphical tool: eye pattern of a 16QAM link.

### System-level modeling strategies in VHDL

The definition and description of a whole system implies the use of a certain VHDL style and the adoption of a modeling methodology to improve the performance of the models without sacrificing their accuracy [2,3]. Among these style details, we would highlight some general ideas:

\* We tried to use the highest possible level of abstraction; remember that we weren't interested on the actual implementation of the blocks, but on their behavior, on their effect on the signal across the whole communication link, so our models were to be a set of algorithms that transformed the signal according to some rules. For instance, we didn't care about how a carrier recovery device works, or about how it is built; we just knew that such a block is able to turn the received constellation and

to align it with the reference axis, so this high-level algorithm was what had to be modelled (fig. 4).

\* Also, we did not pay much attention to clocking; we either defined a simulation clock to control the sequence of events or let the processes execute asynchronously, whenever their input signals changed.

\* A different entity was made for each block in the radio link (see fig. 1 again), as well as a structural architecture for the complete system. This approach proved to be very flexible, as new blocks, like time or phase distortions, could be gradually added to complete the description of the system; furthermore, new architectures for an entity were provided when they were needed (in a project like this, a digital filter can model our channel during the first steps of the design, but it might be necessary to replace it with a more precise time-variant Rummier channel some time later...); finally, we, as designers, were protected against changes in the specification of the system (replacing the original 4PSK modulator by a 16QAM, for instance, is much easier if the modulator is a separated entity than if it is confused inside a bigger block).

\* We made an effort to decide typing soon, and to use high-level types without any clear implication on hardware. In our system, for example, we considered two possibilities: either to create a complex type and to define required operators (adding complex signals means adding their

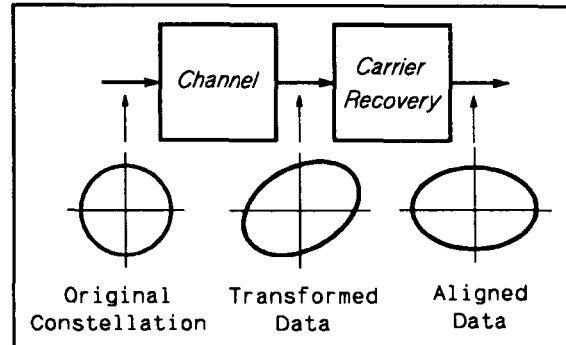


Fig. 4. Effect of the Carrier Recovery block.

two components separately, and so on) or to use the standard real type, taking care of each operation when it was encoded. Records can be used to implement the first alternative:

```
type COMPLEX is record
  R: real;
  I: real;
end record;
```

\* A solution for the lack of mathematical functions in standard VHDL had to be found at the very early steps in the project. As we didn't have a commercial one available, we started building ourselves a math library, including some trigono-

metrical functions which were developed as Taylor series. Some other functions were implemented as tables of values, as their input sets were relatively small. An example of the use of a constant to build the raised cosine function as a table of values is given below:

```
--for filtering purposes, the raised cosine
-- of ALPHA = -3.5T, -3T, -2.5T, ..., 3.5T
-- (just 15 points) is required.
type RCOS_TBL is array (-7 to 7) of real;
constant RAISED_COSINE: RCOS_TBL :=
(0.0057, 0.0000, 0.0171, 0.0000, -0.1200,
 0.0000, 0.0600, 1.0000, 0.0600, 0.0000,
-0.1200, 0.0000, 0.0171, 0.0000, 0.0057);

i := real_to_integer (ALPHA*2/T);
A := RAISED_COSINE (i);
```

```
----- eqadap.vhdl -----
library MGC_PORTABLE;
use MGC_PORTABLE.QSIM_LOGIC.all;

entity EQADAP is
  generic (LENGTH: integer := 4);
  port (CKsymb, RST: in qsim_state;
        TRAINr, TRAINi, --training sequence
        DATA_INr, DATA_INi: in real;
        DATA_OUTr, DATA_OUTi: out real);
  --decided symbols
end EQADAP;

architecture LMS of EQADAP is
  type REAL_VECTOR is
    array (1 to LENGTH) of real;
  signal DLY_INr, DLY_INi: REAL_VECTOR;
  --input data to the linear eq.
  signal Hr, Hi: REAL_VECTOR; --taps
  signal DOUTr, DOUTi: real; --outputs
  signal GHOST: qsim_state := '0';
  constant BURST: integer := 4000;
  constant SINGLE_WORD: integer := 8;
  signal NoSYMB: integer := 1;
  constant A: real := 0.5;
  --step for LMS algorithm
begin
  GHOST <= '1';
  INITIALIZE_COEFF: process (GHOST)
  begin
    for i in Hr'range loop
      Hr(i) <= 0.0;
      Hi(i) <= 0.0;
    end loop;
    Hr(LENGTH/2) <= 1.0;
    Hi(LENGTH/2) <= 1.0;
  end process INITIALIZE_COEFF;

  SHIFT_IN: process (CKsymb, RST)
  begin
    if RST = '0' then
      for i in DLY_INr'range loop
        DLY_INr(i) <= 0.0;
        DLY_INi(i) <= 0.0;
      end loop;
    elsif CKsymb'event and CKsymb = '1' then
      for i in LENGTH downto 2 loop
        DLY_INr(i) <= DLY_INr(i-1);
        DLY_INi(i) <= DLY_INi(i-1);
      end loop;
      DLY_INr(1) <= DATA_INr;
      DLY_INi(1) <= DATA_INi;
    end if;
  end process SHIFT_IN;

  EQ: process (CKsymb)
  variable ACUMr, ACUMi: real;
  variable ERRORr, ERRORi: real;

  begin
    if CKsymb'event and CKsymb = '1' then
      --1) linear filter:
      ACUMr := 0.0; ACUMi := 0.0;
      for i in DLY_INr'range loop
        ACUMr := DLY_INr(i) * Hr(i) + ACUMr;
        ACUMi := DLY_INi(i) * Hi(i) + ACUMi;
      end loop;

      --2) decision device:
      if (ACUMr > 0.0) then
        if (ACUMr > 2.0) then
          DOUTr <= 3.0;
        else DOUTr <= 1.0; end if;
      else
        if (ACUMr < -2.0) then
          DOUTr <= -3.0;
        else DOUTr <= -1.0; end if;
      end if;

      if (ACUMi > 0.0) then
        if (ACUMi > 2.0) then
          DOUTi <= 3.0;
        else DOUTi <= 1.0; end if;
      else
        if (ACUMi < -2.0) then
          DOUTi <= -3.0;
        else DOUTi <= -1.0; end if;
      end if;

      --3) Error:
      if (NoSYMB <= SINGLE_WORD)
        ERRORr := TRAINr - ACUMr;
        ERRORi := TRAINi - ACUMi;
      else
        ERRORr := DOUTr - ACUMr;
        ERRORi := DOUTi - ACUMi;
      end if;
      if (NoSYMB = BURST) then
        NoSYMB <= 0;
      else
        NoSYMB <= NoSYMB + 1;
      end if;

      --4) LMS algorithm:
      for i in Hr'range loop
        Hr(i) <= Hr(i) + A *
          (ERRORr * DLY_INr +
           ERRORi * DLY_INi);
        Hi(i) <= Hi(i) + A *
          (ERRORr * DLY_INr -
           ERRORi * DLY_INi);
      end loop;
    end if;
  end process EQ;

  DATA_OUTr <= DOUTr;
  DATA_OUTi <= DOUTi;
end LMS;
```

Fig. 5. High-level description of the equalizer.

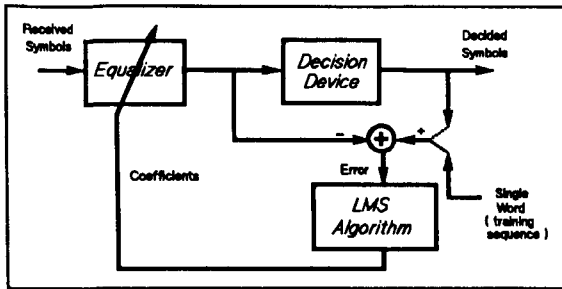


Fig. 6. Structure of the adaptive equalizer.

## The selected equalizer. VHDL model

After trying several architectures for our receiver, including linear, adaptive, fractionally-spaced, blind, complex and decision-feedback equalizers of different lengths, and after studying both Least Mean-Square (LMS) and Recursive Least Squares (RLS) algorithms to recalculate the coefficients, a complex LMS-adaptive symbol-spaced 4-tap equalizer was chosen (see figures 5 and 6).

The entity of our equalizer includes one generic and eight ports: the only generic, LENGTH, provides a useful mechanism for the definition of the number of coefficients in our device, while the ports are a simulation reset and a simulation clock (RST and CKsymb), two inputs for the training sequence (TRAINr, TRAINi) and the input lines for the received data and the output lines for the equalized symbols.

When describing the behavior of the device, some modeling details appeared to be specially relevant (comments will always be referred to fig. 5):

\* First of all, we defined which signals were required (because they had to be saved for the future, for instance). The remaining values were kept in variables, which simulate a lot faster. Sometimes, no additional signals had to be defined inside an architecture, which could then be described as just a single process. In our equalizer, just the coefficients (H) and previous inputs (DLY\_IN) had to be signals, so two processes were enough: SHIFT\_IN moves the input data into DLY\_IN and EQ recalculates H and computes the output symbols.

\* Our processes are controlled with a simulation clock, CKsymb. A simulation reset (RST) can also be

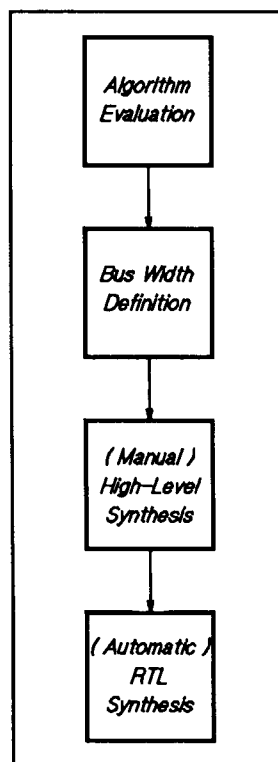


Fig. 7. Design flow.

added, if it is of any use.

\* As figure 5 shows, we decided to represent our data as couples of real-valued signals, so operations with them had to be instantiated very carefully (see an example of complex signals substractions under '--3', and a complex product following '--4'. These operations are simply

```
ERROR := TRAIN - ACUM
```

and

```
H <= H + A * ERROR * DLY_IN
```

where ERROR, H, TRAIN, ACUM and DLY\_IN are complex-typed signals, and A is a real value).

\* It must be remembered that CKsymb is a simulation clock, and that it can be used with as much freedom as needed. For instance, although our final solution was a non-fractionary filter, we studied a fractionary approach, where the input data was shifted twice in each clock period; a sentence like

```
elsif CKsymb'event then
```

was then used, even though it is clear that such a construction would never appear in a register transfer-level description.

\* An extra process (with the very descriptive name of INITIALIZE\_COEFF) was created to initialize the arrays of variable length that must start with a certain value (H, in this case). While the other signals can be preloaded with a value at declaration time (see signal NoSYMB), arrays of unknown length require the use of a loop inside a process that is executed only once, at the beginning of the simulation time. This can be arranged with a GHOST signal that is assigned a certain value at time zero, implying a change that triggers the initialization process. Of course, the GHOST signal cannot be used any more, to prevent unexpected re-initializations of the arrays.

## The path from high-level VHDL to the final implementation

Even though our contribution was focused on high-level design, we would not like to come to an end without giving some ideas about the future steps in the project. An in-depth study

of these steps for a similar application, anyhow, can be found in [4].

At this point, many different types of equalizers have been studied, one of them has been chosen as the most suitable for our application, and some others might still remain as good alternatives if the selected architecture has to be dropped for any reason. This is the time, then, to try to find a hardware implementation for our algorithm. This path from the high-level definition of an algorithm to the design of an integrated circuit implementing such mathematical operation can be separated into three main phases, shown on fig. 7.

First of all, the bus width has to be defined. As, until now, we had not taken care of the precision required to implement our architectures, it may happen that the one which used to have the best performance can only work with impractical bus sizes, so other alternatives should be re-simulated before a final choice is made.

As the synthesis tools available today are still unable to take care of some complex tasks, like resource assignment, scheduling or pipelining, part of the work has to be made by hand. This second phase will end with a register transfer-level description of the equalizer that can be automatically synthesized during the third and last step into a net list of gates.

## Conclusions

A system-level modeling experience in VHDL has been presented, paying special attention to the VHDL style and modeling strategies required for the high level of abstraction of such applications. A brief chapter was also dedicated to the tools needed for a complex project, discussing their capabilities and detailing the improvements needed to analyze the radio link that was under study.

## References

- [1] S. Qureshi, "Adaptive Equalization", Proceedings of the IEEE, Sep. 1985.
- [2] C. Long, "System Design Methodology Using VHDL", VHDL Users' Group Spring 1991 Conference.
- [3] A. Sama, J. Armstrong, "Behavioral Modeling of RF Systems with VHDL", VHDL Users' Group Spring 1991 Conference.
- [4] P. Scheidt, "Digital Filter Design Using VHDL", Synopsys Methodology Notes, Aug. 1992.
- [5] "IEEE Standard VHDL Language Reference Manual", IEEE, 1988.