INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Tools for Correct DSP Synchronization

Alain Kerihuel, Roderick McConnell, Frédéric Raimbault

N° 1973

Avril 1993

PROGRAMME 1

Architectures parallèles,
bases de données,
réseaux et systèmes distribués

INRIA
RENNES

Tools for Correct DSP Synchronization*

Alain Kerihuel, Roderick McConnell, Frédéric Raimbault **

Programme 1 — Architectures parallèles, bases de données, réseaux et systèmes distribués
Projet API

Rapport de recherche n° 1973 — Avril 1993 — 11 pages

Abstract: Software tools for assuring proper synchronization of real-time digital systems targeted at high-throughput applications, are presented. These tools depend on the SIGNAL synchronous language to express timing relations. The tools are presented in the context of a data-flow environment. Tools and environment support the use of externally defined functional blocks, for rapid prototyping and easy insertion of new models of those functional blocks. We present 34 Mbit/sec video coding as a sample application using our tools.

Key-words: digital signal processing, specialized systems design, synchronous language

(Résumé : tsvp)

Soumis à 1993 IEEE Workshop on VLSI Signal Processing

*This work was partially funded by the French Coordinated Research Program ANM of the French Ministry of Research and Space and by the Esprit BRA project No 6632 NANA-2.

**{Alain.Kerihuel}{Roderick.McConnell}{Frederic.Raimbault}@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 38 38 32

Des outils pour la synchronisation des systèmes traitement digital des signaux

Résumé : Nous présentons des outils logiciels pour assurer la bonne synchronisation des systèmes digitaux temps-réel, ciblés vers des applications à haut-débits. Leurs définitions s'appuient sur le langage synchrone SIGNAL pour exprimer les relations temporelles. Les outils sont présentés dans le contexte d'un environnement flot de données. Des opérateurs fonctionnels externes peuvent être utilisés afin d'accélérer le prototypage et d'en faciliter la maintenance. L'exemple qui sert de support à notre méthodologie est celui de la chaîne de codage vidéo 34 Mégabit/s.

Mots-clé : conception de systèmes spécialisés, langage synchrone, traitement digital de signaux

1 Introduction

High-throughput Digital Signal Processing (DSP) systems present an ideal opportunity for the application of Correct-by-design techniques. The algorithms used are typically highly regular, and based on a sequence of simple operations. For real-time DSP processing, we propose a technique and tools which allow the designer to simulate solutions at different levels, from abstract data-flow paradigm to concrete models of circuits and dedicated processors.

Our technique uses tools written in SIGNAL, a synchronous data-flow programming language, together with C-based imperative languages. SIGNAL allows us to express the instants at which a particular data value exists, and at a higher level to assure the correct interaction of the complete system. It also provides a vehicle for introducing correct transformations on communications domains. The imperative languages are used for expressing low-level operations.

VLSI components, whether programmable processors or dedicated circuits, are essential to modern digital hardware design. Thus we demand that our methodolgy be capable of elegantly representing both programmable processors and specialized circuits. Our technique depends heavily on the idea of "functional blocks", operators which perform one particular operation of an algorithm. A complex system is built up using functional blocks, either pre-existing (such as commercial VLSI circuits) or particular to the problem at hand (programs for a SIMD array). In the case of high-throughput synchronous processing, such as in real-time video processing, these blocks must operate in parallel to perform the necessary operations at the required speed.

A major limitation of our approach, and also what we consider to be one of its strengths, is a decision to consider only Synchronous Data Flow, as defined by Lee et al. [11, 12]. In Synchronous Data Flow, the functional blocks have a fixed ratio between input and output. This vision is more limited than the traditional version of data-flow, but it offers the advantage of a simple model for the synchronization between functional blocks. Our technique thus applies to data-flow graphs with no conditional links (no external data-dependent branching), such as can be found in real-time video processing or other high-throughput applications. However, the same limitation makes it easy to model the control of our systems using SIGNAL, and hence to be able to explicitly check that communications are well synchronized. We note that the concept of Synchronous Data Flow has been realised in several simulation environments from Berkely, the most recent being PTOLEMY [1], but without the support of a synchronous language.

The remainder of this article presents the languages of our environment in section 2, an overview of our method in section 3, a demonstration of our technique using the application video coding at 34 Megabits/sec [5] in section 4, and finally a more detailed look at the tools we have developed in section 5, using the example from section 4.

2 The Languages

Our environment uses two different categories of langages, data-flow and imperative, targeted at different aspects of modelisation. Each category of language is suited for its specific domain, but not necessarily elegant in expressing all aspects of a DSP system.

To coordinate operations of functional blocks at the system level, we use SIGNAL. SIGNAL is a synchronous language, and thus allows us to explicitly express the presence or absence of a value at a particular instant. Once the control has been programmed in SIGNAL, a check can be performed to assure the proper synchronization of control operations. A graphics interface, XSIGNAL, allows the designer to visualize the complete system, and the interaction between various functional blocks. The language SIGNAL is described in greater detail in the first of the following subsections. The tools which we have developed using SIGNAL are presented in section 3.

While SIGNAL allows us to easily express synchronization of data transfers, we choose to use C [8] for expressing the operations of a functional block. It has been our observation that a fair number of VLSI circuits and dedicated systems are modeled in C during their development, and we believe it is important

to maintain compatibility with this de-facto standard. The C language is also well-supported in the UNIX environment, our environment of choice.

In an attempt to bridge the gap between C code in a UNIX environment, and regular VLSI architectures, we use RELACS. The RELACS language is a close cousin to C, with extensions to support programming a synchronous regular array of processors. We use RELACS to model high-throughput functional blocks, as these are obvious targets for VLSI implementations. This language is described in the second subsection.

To simulate the operation of our models, we use C code generated by the RELACS and SIGNAL compilers, together with functional blocks and sub-systems written directly in C. In section 4 we present a model of 34 Mbit/sec coding, to demonstrate how the different languages are used together, and to show how the tools written in SIGNAL can be used to assure the syncronization of communications.

2.1 The SIGNAL language

SIGNAL[10] is a synchronous data-flow language for the conception of real-time Digital Signal Processing and control applications. SIGNAL expresses operations on an infinite series of values, called a *signal*. A *signal* is composed of a couple (stream, clock), where the stream is an infinite ordered sequence of typed values, and the clock specifies the instants when the values are present.

A SIGNAL program describes temporal, as well as functional, relations between operations. We call a SIGNAL program a "process", because it operates in a data-flow fashion [3], possibly in parallel with other processes.

2.1.1 Operators on signals

The SIGNAL kernel is composed of four operators which manipulate **signals**; they are the elementary processes used to composed a program. The four operators are as follows:

- The extension of the classical operators (+,*,...) to synchronous signals.
  z:= (x *op* y) defines an elementary process such that $\forall t \geq 1\ z_t := (x_t\ op\ y_t)$, where x_t and y_t are signals, and *op* is a binary function on signals.

- A dynamic instruction, the "$' or delay. The "$" behaves like a FIFO of a specified length. Thus $zx := x\$1$ significates that $\forall t : zx_t := x_{t-1}$ with x initialized to x_0.

- The **when** operator is a conditional under-sampling. $x := y$ **when** c means that the signal x takes the value of the signal y when the value of the boolean signal c is true.

- The **default** operator performs over-sampling. $x := u$ **default** v means the values of the signal x are those of the signal u, when they exist, or else those of signal v.

The first two operators presented above are *monochronous*, meaning that their signals are synchronous (occur at the same instant). The **when** and **default** operators are *polychronous* i.e. their signals may have different clocks. All these operators, with the exception of the delay $, are static operators, meaning that the output depends only on the inputs at the instants when the inputs are available, independent of past values (one can draw an analogy with combinatorial logic). Static operators present an output at the same instant as the inputs are valid; hence, there is no concept of duration of an operation in SIGNAL. The $ is qualified as dynamic because it takes into account the past.

- the "|" operator or composition operator:
  The operators of the SIGNAL kernel are elementary processes that are composed to build a SIGNAL program. For example, $P \mid Q$ is the parallel composition of two elementary instructions or processes P and Q. Parallel processes communicate via shared signals.

```
process COUNTER=
(integer PERIOD)
{ ?  logical IN
! integer RN}
( |  RN  := (1 when not IN) default (1 when RZN = PERIOD) default RZN+1
|  RZN  := RN $1
|  synchro {RN, IN}
| )
where
integer RZN init PERIOD
end
```

Figure 1: A counter in SIGNAL

2.1.2 SIGNAL extensions

The language SIGNAL also provides the programmer with other instructions built on SIGNAL-kernel operators. The most important are the **synchro** $\{ x_1, \ldots, x_n \}$, which permits the programmer to synchronize signals with each other, and the operator **cell**, which allows the memorization of a value.

To help the designer describe his or her application, external processes, written in "C" for example, can be called from SIGNAL. The interface between external processes and SIGNAL is synchronous, meaning that all inputs and outputs have an event (occur) at the same instant. A SIGNAL program is described with the help of an graphical environment, XSIGNAL which supports modular and hierarchical programming. Examples of this environment will be given in the following sections.

The SIGNAL environment also provides the programmer with tools to verify certain properties regarding the synchronization of signals. A SIGNAL program is expressed in the finite field $Z/3Z$ of integer modulo 3, and under certain conditions, the systems of equations implied by a SIGNAL program can be solved by means of these tools[4].

2.1.3 An example: a modulo counter

The SIGNAL process given in (figure 1) describes a counter modulo on a logical clock signal IN. The events of the input signal when IN is true are counted modulo PERIOD, the period of the counter. In the example the input signal IN is a logical one. The ouput integer signal RN counts the number of successive events of IN with the value "true" . The signal RN takes the value one each time that the signal is "false" (1 **when not** IN) and if the condition is not verified on the signal IN, each time the counter is equal to PERIOD. Otherwise, the signal RN is equal to the value of the signal RZN - the last value of the signal RN - plus one. The **default** operator is an union of events; the first value is given precedence if both values have an event at the same instant. RN and IN are synchronized with the **synchro** operator; all events of RN and IN are completely synchronous. The local integer signal RZN is initialized with the value PERIOD to start the counter.

2.2 The RELACS language

To support efficient systolic experiments on programmable parallel architectures, one of the authors (F. Raimbault) has developed RELACS. The RELACS compiler generates code which will perform a required function on a regular processor array, and can also be used to model VLSI circuits. This language embodies the computation and the communication aspects of systolic algorithms in a terse programming model, with a syntax close to that of the C language.

The target architecture consist of a network of multiple identical cells running in SIMD mode (all the cells are doing the same thing at the same time). Data management outside the array is also taken into consideration and is viewed as an independent process. This systolic architecture is based on a programmable systolic machine called MICMACS [6, 9]. The user views the MICMACS system as a programmable accelerator connected to a general purpose UNIX host workstation.

The user writes a single source program, from which the compiler generates code for execution both on the host and on each cell of the network. Partitionning of the code is done by data types. The user is given

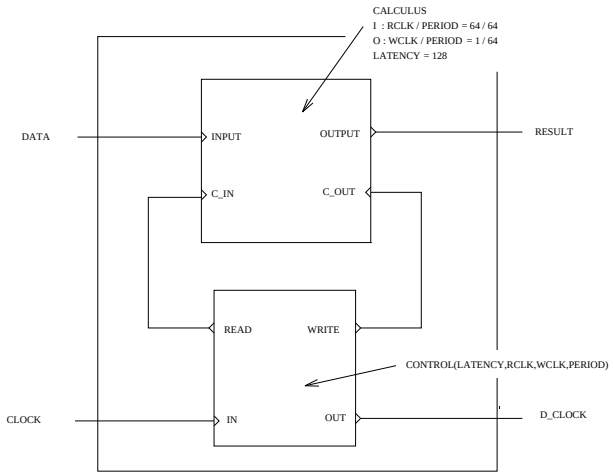

Figure 2: A generic description for a functional block in SIGNAL

full control over computation partitioning and algorithm design. The compiler is in charge of exploiting parallelism at the system level and at the cell level. The compiler also handles the details of communication protocols between each components and code generation for the systolic processors.

In order to integrate RELACS functional blocks into our environment, we use C code generated by the RELACS compiler for a sequential target. The code is linked together with C code generated by the SIGNAL compiler, and other sub-systems written in C, to create an executable model under UNIX.

3 Modeling an application using SIGNAL

An application is decribed as a connected set of fuctional blocks. It is typically deduced directly from the block diagrams used to represent DSP applications at a high level. Each block repeats a specific task, with a certain period. A block is caracterized by a certain latency (duration of the operation), and the input and output events during one period. To manage a block in a SIGNAL description, special processes generate the control operations for each input and output signal of a block. These processes assure that there is the proper number of input and output signals per period, and also calculate the correct delays due to the latency of the block. A sample generic description of a block is given in (Figure 2). Here CLOCK gives a period of 64 clock events (clock events correspond to clock cycles, not to be confused with the period), and the operation has a latency of 128 clock events. The output signal "OUT" of the control block provides a clock synchronized with the output of the operator, to be used as the input to a successor. In this case the signal "OUT" will be the signal "IN", delayed by 128 clock events - the latency of the block. The outputs READ and WRITE are respectively the signals which manage the number RCLK of inputs, and the number WCLK of outputs of the operator; for the given example, an input for each clock event and one output per 64 events of the clock.

4 Control and Synchronization at 34 Mbit/s

The 34 Megabit per second video compression standard is used in the television industry for the transfer of program material[5]. This standard is particularly interesting to us because commercial products exist, both for the functional blocks and as complete systems. These serve as benchmarks, to assure that our theories reflect the actual state-of-the-art in hardware. A schematic for 34 Megabit/second image coding is presented

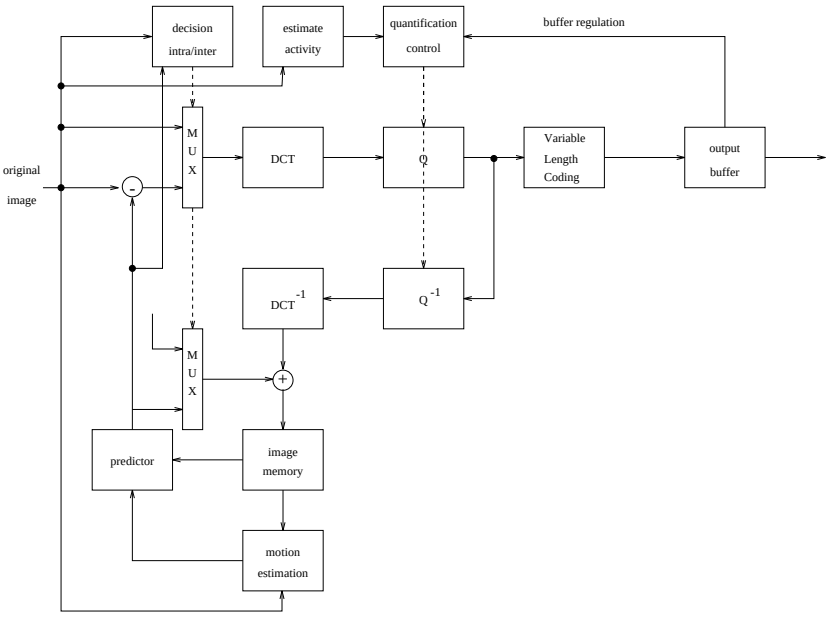

Figure 3: Structure of the 34 Megabits/sec Coder

in (Figure 3)[2]. It compresses images using motion estimation, and a Discrete Cosine Transform (DCT) followed by variable-length encoding. The same schema is used for the graphical interface XSIGNAL to the Synchronous Data-Flow language SIGNAL.

The complete coder is described in 300 lines of SIGNAL for the control, and 2000 lines of RELACS and C for the functional blocks (not including the code to do I/O on a sequence of images). We will not attempt to present the whole of this standard, but rather will extract a subset of the coder which performs image compression using a DCT and quantification of the resulting coefficients. This is presented in the following subsection.

4.1 The Mini-coder

Image compression within an image (*intra-image* compression) is performed using the sequence of operations DCT, quantification, variable length coding. We will use this subset to present in greater detail the different tools. The sequence, as coded in SIGNAL under the XSIGNAL interface, is presented in (Figure 4).

4.1.1 The DCT functional block

RELACS was chosen to model the DCT operation for simulation. It allows us to generate code targeted at a systolic parallel processor, but which can also be executed on a workstation under UNIX. In order to have a more realistic model, we implement the fixed-point operations of the inmos/SGS-THOMPSON IMS-A121 8x8 DCT chip [7]. The transform performed by this circuit uses two one-dimensional cosine transforms, as given by:

$$X(k) = \sqrt{2/N}c(k)\sum_{i=0}^{N-1} x(i)\ cos\frac{(2i+1)k\pi}{2N} k = 0, 1, \ldots, N-1 \quad \text{where}$$

$$c(k) = \begin{cases} \frac{1}{\sqrt{2}} & for \quad \text{k=0} \\ 1 & for \quad \text{k=1,...,N-1} \end{cases}$$

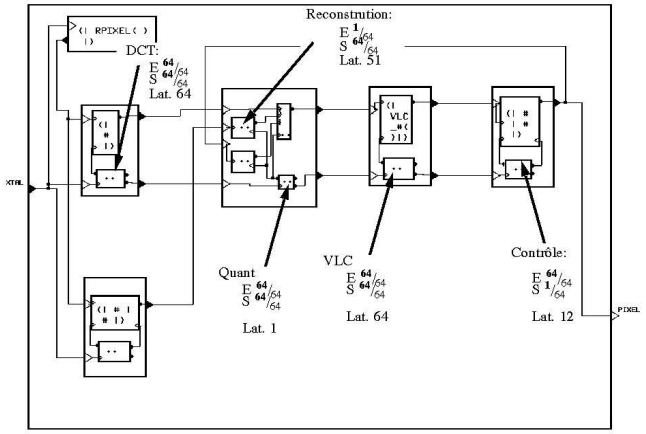

Figure 4: Image compression with DCT

Our model implements multiplication, saturation, and rounding operations to the same number of bits as the circuit, with the exception of the final matrix multiply, where we pass only 32 bits instead of 33. The code assumes 32 bit unsigned integers.

4.1.2 Control for the DCT functional block

With each functional block in the coder, we associate a control block, written in SIGNAL. The control blocks use the code given in (Figure 6). In (Figure 4) these are the small boxes beneath the "functional blocks". In the case of the DCT functional block, if we choose the specifications of the IMS-A121, this processor reads one image pixel per clock, and writes one transformed coefficient per clock, with a latency of 128 cycles. Thus the control parameters are: LATENCY = 128; RCLK = WCLK = 64; and PERIOD = 64.

4.2 Assuring Correct Synchronization

The resynchronization block, as presented in section 3, is used to assure that the communication in the coder is correctly synchronized during simulation. In the following subsections, we present a simple control-loop communication found in the mini-coder, and then we present the SIGNAL blocks used to insure that this communication is correctly sychronized.

4.2.1 The Communication

As data passes through the coder, following the cosine transform the 8x8 blocks of DCT coefficients are quantized. Quantization is based on two control parameters - an estimate of the activity in a particular 8x8 block, and a buffer occupancy factor. The activity factor is calculated once per block, while the buffer occupancy factor is calculated once per stripe, i.e. a one-block-high horizontal slice of the image. To simplify our model, we will consider only the buffer occupancy factor, and consider that a new value is calculated for each block.

The control challenge is to insure that the new value for the buffer occupancy factor 'f' is correctly synchronized with the start of a block. This problem does not exist in the block-level model of the system, as the block and the factor f are simultaneous, or in the notation of SIGNAL, *synchro* {block, f}. However, when one descends to an implementation level, where 8 x 8 blocks are transferred as a series of pixels (as they would be if taken from the output of the IMS-A121 DCT chip), it becomes necessary to synchronize f with the start of a block.

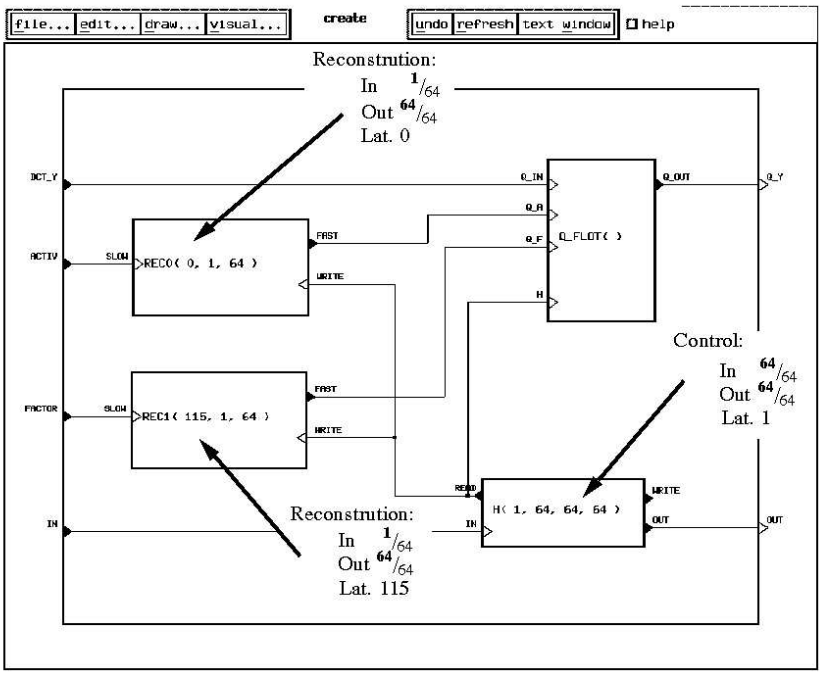

Figure 5: Assuring Synchronization of 'FACTOR'

4.2.2 The Synchronization

The solution to the synchronization problem is intuitively obvious: it suffices to latch the new value for f when it arrives, and present it to the quantifier simultaneous with the arrival of the first pixel in a block. In order to be able to assure the correctness of the synchronization during simulation, we express this property in SIGNAL. In (Figure 5) the section of the coder which performs this check is shown.

The block 'Rec1', which implements the code of (Figure 5), performs the resynchronization of the occupancy factor. It takes the parameters Latency, Read-cycle, and Write-cycle. In this example, correct synchronization demands that the new factor be delayed by 115 cycles to be synchronized with the start of a new block, and that it be sampled 64 times. Thus in this case the parameters are (115, 1, 64). If the factor f is not correctly synchronized, an error results during simulation.

5 Control and Synchronization tools

We have developed some generic tools in order to help the designer realize an application. These tools are parametrized processes written in SIGNAL. Two of them are presented in this section. One is a control "black box" that is used to provide the control for a functional block, and the other a process that will resynchronize signals which arrive at regular intervals and assure that the synchronization is correct.

5.1 Generic control "black box"

A process which describes a generic control "black box" for a synchronous functional block is given in (Figure 6). This process allows the specification of output events based on input events. We note that the synchronous hypothesis allows us to calculate all input and output events based on one input, provided this input has an appropriate clock.

```
process H=
(integer LATENCY, RCLK, WCLK, PERIOD)
%  the parameters are:  %
%   LATENCY - the number of clock cycles between IN true, and OUT true   %
%  RCLK - the number of clock cycles that READ is true per period   %
%   WCLK - the number of clock cycles that WRITE is true per period   %
%   PERIOD - the number of clock cycles in a period   %
%   This code provides a generic version of a control "black box" for doing development   %
{ ?  logical IN
!  logical READ;
logical WRITE;
logical OUT init[ {to LATENCY} : false] }
( |  % all signals are synchronous with the input in  %
synchro {IN, OUT, RN, WN, READ, WRITE}
|  OUT := in $LATENCY
% a modulo counter for READ and WRITE          %
|  RN := (1 when (not IN) )default (1 when (RZN=PERIOD) )
default (RZN+1)
|  WN := (1 when (not OUT) )default (1 when (WZN=PERIOD) )
default (WZN+1)
|  READ := (true when( (RZN<=RCLK) when IN) )default false
|  WRITE := (true when( (WZN<=WCLK) when OUT) )default false
|  RZN := RN $1
|  WZN := WN $1
| )
where
integer RN, RZN init PERIOD;
integer WN, WZN init PERIOD
end
```

Figure 6: Control clock calculus program in SIGNAL

The parameters to the control process allow us to specify the number of cycles a period has, and to give a latency to an input clock signal. These must correspond to the caracteristics of the functional block. The READ and WRITE signals command the input and output of a fuctional block. The number of input and output events in one cycle of the functional block are given by the RCLK and WCLK parameters. Here the process H has one boolean input signal IN, and three boolean output signals READ,WRITE and OUT. LATENCY, RCLK and WCLK are the parameters to the process. By specifying **logical** OUT **init**[**to** LATENCY : false] , the output signal OUT is initialized to false during its first LATENCY instances. **synchro** {IN,OUT,RN,READ,WRITE} synchronizes all the concerned signals with the input clock signal IN. The RN and WN signals are two counters modulo PERIOD like that is defined in the figure 1. The signal RZN<=RCLK is the sequence of the values $RZN_i <= RCLK$ with i an instant where the local signal RZN_i is less or equal than the value of RCLK. In (Figure 6), the values of the signal (RZN <= RCLK) **when** IN are the values of the signal (RZN<=RCLK), when they exist, at the instants where the signal IN is true. Then, for example, the ouput signal READ takes the value "true" each time the number of events of IN is less or equal than PERIOD and that there is an event on IN; else it takes the value "false". The instruction OUT:=IN $LATENCY means that the values of the signal OUT are the values of the signal IN with a delay of LATENCY instants. The four integer local signals RN, RZN, WN, WZN are specified after the **where** declaration. RN and RZN are initialized to the integer value "1".

5.2 Resynchronization process

The process given in (Figure 7) permits the resynchronization of an input signal; it can delay the signal values by a certain number of cycles, and perform over-sampling as needed. At the same time, a test is performed to assure that the resulting signal is properly synchronized with the clock of the functional block. This process is used in conjuction with the control process of a given functional block.

The resynchronization process takes three parameters. The first, LATENCY, is the latency imposed on an input signal to resynchonize it with a given functional block. The RCLK and the WCLK parameters are, respectively, the number of clocks when the input signals 'SLOW', and 'WRITE', are true during one period of the functional block (they are only used to give the relation between the number of inputs and outputs).

Checking the correctness of the synchronzation consists of insuring that there is a proper delay between the instant that the input signal arrives and the moment it is demanded by the control process, and if there is up-sampling, that the control process demands the correct number of values for each input. These

```
process REC1=
(integer LATENCY, RCLK, WCLK)
% the parameters are:                                    %
%                                                       %
%      LATENCY - clocks between SLOW and WRITE %
%      RCLK -    clocks of SLOW per period %
%      WCLK -    clocks of WRITE per period %
%                                                       %
%  This code is intended to be a generic version of a %
% resynchronization "box" with delay; it comprises also the    %
% check of synchronization between the input SLOW and    %
% the clocking of the output by WRITE.                  %
%  Note that RCLK and WCLK are only used to establish a ratio  %
% between the number of input and output clocks per period.    %
%                                                       %
{ ?  integer SLOW;
logical WRITE
!  integer FAST}
( |  X_SLOW := SLOW cell (event WRITE) % up-sample the input %
|  FAST := (X_SLOW $LATENCY) when WRITE    % delay and cadence   %
|  % ===> everything below is for the detection of errors  <===  %
SYNC_ERR := (false when (event FAST) when (H_SLOW $LATENCY) ) % condition of no error %
default (true when (event FAST) )default (true when (H_SLOW $LATENCY) )
|  ZSYNC := SYNC_ERR $1
|  ERR_CNT := # SYNC_ERR after (not SYNC_ERR)  % count the number of cycles of error %
|  ZE := ERR_CNT $1
|  PRINT_CNT := ZE when (ZSYNC and (not SYNC_ERR) ) % write error message once only  %
|  H_SLOW := (true when (event SLOW) )default (true when (N<= (WCLK/RCLK) ))
default (false when (event X_SLOW) )   % a clock & counter for 'good' events %
|  N := # (event X_SLOW) from (event SLOW)
|  PRINT_ERR_CNT ()
| )
where
integer X_SLOW;
logical H_SLOW;
integer N init (WCLK/RCLK) +1;
logical SYNC_ERR, ZSYNC init false;
integer ERR_CNT, PRINT_CNT, ZE init 0
% an external (O/S dependent) call to write an error message with a count %
function PRINT_ERR_CNT=
{ ?  integer PRINT_CNT
! }
end
```

Figure 7: Synchronization check program in SIGNAL

relationships are fixed at the time of compilation, which is possible because we make the assumption that the system is synchronous.

There are two distinct groups of instructions in this process. The first generates the resynchronized signal, and the second handles the check for proper synchronization. There are two input signals, an integer 'SLOW' which is the signal to be resynchronized, and a logical clock signal 'WRITE', which is provided by the control process. The expression X_SLOW:=SLOW **cell** (**event** WRITE) memorizes the current value of SLOW, and also performs an over-sampling operation for each event of the WRITE signal. This value is delayed by a certain latency to complete the resynchronization, through the expression FAST := (X_SLOW $ LATENCY) **when** WRITE. Some of the check code deserves a fuller explaination as well. The #(**event** X_SLOW)**from**(**event** SLOW) and #SYNC_ERR **after** (**not** SYNC_ERR) instructions use predefined counters. The first is used to insure the proper ratio of events between incoming and outgoing signals; the second to count error events. The function PRINT_ERR_CNT is an external call used to print error messages. For our purposes it is written in "C" for the UNIX operating system.

6 Conclusion

This article presents tools to model synchronous digital systems for high-throughput applications. Particular to our method is the symbiosis of a synchronous programming language to assure the validity of the control, and languages dedicated to highly-regular architectures. We restrict our efforts to synchronous DSP systems, where the algorithms are typically simple and highly regular, and the synchronization between functional blocks can be easily modeled. The environment relies heavily on a data-flow paradigm, and includes tools to check the synchronization of data transfers in a data-flow model.

The tools for control and resynchronization are completely synchronous, which permits us to express the same control function in terms of equations for programmable (PAL) circuits. For example the control process fits into one or two 22V10 PAL's, depending on the size of the modulo counters needed. The resynchronization

process can also be expressed in terms of equations for PAL's. However, the hardware representation no longer performs the check for synchronization errors, as it is assumed that synchronization errors will have been detected during simulation.

The 34 Mbit/sec video coding algorithm was used as a demonstration of an application in this environment. This standard is particularly interesting to us because the realization of coders is well understood, and commercial products exist. An important consideration for us was that our methodology and representation reflect actual state-of-the-art hardware. Thus, for example, the 34 Megabit standard requires a discrete cosine transformation, but does not specify how this be implemented; the design environment presented allows one to create encapsulated models of different DCT circuits, and to simulate a system using these different versions.

The tools presented, which check that data transfers are correctly synchronized during simulation, are written in SIGNAL. Recent work in static analysis of SIGNAL programs makes it possible to perform some of this checking as verification without simulation. Using a subset of the mini-coder in (Figure 4), it was possible to prove that the system cannot enter into the state where the value f is not correctly synchronized with the first pixel of an 8 x 8 block. It is our hope that in the future this static verification approach can be used to prove correct synchronization in entire systems.

7 Thanks

Our thanks to Bruno Dutertre, who took the time to pass what he could of the mini-chain through his static analysis tools.

References

[1] Joseph Buck, Edwin Goei, Soonhoi Ha, Ichiro Kuroda, Phil Lapsley, Edward Lee, and David Messerschmitt. *The Almagest.* Technical Report, University of California, Berkely, Mar 1991.

[2] F. Charot. Application de codage d'image dct à 34 mégabit/s. Apr 1990. *unpublished.*

[3] J.B. Dennis, J.B. Fosseen, and J.P. Linderman. Data flow schemas. *Lecture Notes in Computer Science*, 5:187–216.

[4] Bruno Dutertre. *Spécification et preuve de systèmes dynamiques.* PhD thesis, Université de Rennes 1, FRANCE, Dec 1992.

[5] ETSI. *Codage d'image DCT 34 Mégabit/s.* Technical Report, European Telecommunication Standard Institute, Dec 1990.

[6] P. Frison, D. Lavenier, H. Leverge, and P. Quinton. A vlsi programmable systolic architecture. In *International Conference on Systolic Array*, 1989.

[7] inmos. Ims a121 2-d discrete cosine transform image processor. Data Sheet, 1991.

[8] Kernighan, W. Brian, and D. M. Ritchie. *The C programming language.* Prentice-Hall, 1978.

[9] Dominique Lavenier. MicMacs : un réseau systolique linéaire programmable pour le traitement des chaines de caractères. Thèse de l'Université de Rennes 1, jun 1989.

[10] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, apr 1991.

[11] Edward A. Lee and David G. Messerchmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEETC*, C-36(1), Jan 1987.

[12] Edward A. Lee and David G. Messerchmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9), Sep 1987.

Unité de recherche INRIA Lorraine, Technôpole de Nancy-Brabois, Campus scientifique,
615 rue de Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399