



Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON-UCBL n° 5668

***Cycle Accurate Simulation Model
Generation for SoC Prototyping***

Antoine Fraboulet
Tanguy Risset
Antoine Scherrer

Mai 2004

Research Report N° 2004-18

École Normale Supérieure de Lyon

46 Allée d'Italie, 69364 Lyon Cedex 07, France

Téléphone : +33(0)4.72.72.80.37

Télécopieur : +33(0)4.72.72.80.80

Adresse électronique : lip@ens-lyon.fr

Cycle Accurate Simulation Model Generation for SoC Prototyping

Antoine Fraboulet
Tanguy Risset
Antoine Scherrer

Mai 2004

Abstract

We present new results concerning the integration of high level designed IPs into a complete System on Chip. We first introduce a new computation model that can be used for cycle accurate simulation of register transfer level synthesized hardware. Then we provide simulation of a SoC integrating a data-flow IP synthesized with MMAAlpha and the SocLib cycle accurate simulation environment. This integration also validates an efficient generic interface mechanism for data-flow IPs.

Keywords: system on chip, high level synthesis, VLSI, circuit simulation, computer aided design tools

Résumé

Nous présentons des résultats utiles à l'intégration de composants matériels générés par synthèse de haut niveau dans un système sur puce. Nous introduisons d'abord une nouvelle manière de modéliser les descriptions de circuit de niveau transfert de registre qui peut être utilisée pour créer des modèle de simulation précis au cycle et au bit près. Nous expérimentons ensuite la simulation d'un système sur puce complet intégrant des composant de la bibliothèque SocLib et de l'outils de synthèse MMAAlpha. L'intégration valide aussi le mécanisme d'interface générique que nous avons proposé pour les architectures générées avec MMAAlpha

Mots-clés: système sur puce, synthèse de haut niveau, simulation de circuits, VLSI, conception de circuits

Contents

1	Introduction	2
2	FSM representation for RTL simulation models	3
2.1	A new FSM model for hardware modelling	4
2.2	SystemC implementation	8
3	SoC platform simulation	10
3.1	Generic interface for stream processing accelerators	10
3.2	Experimental platform setting	13
3.2.1	SocLib and MMAAlpha	13
3.2.2	The SOC platform	14
3.3	System and simulation performances	15
4	Conclusion	16
A	New FSM model for one cell of the DLMS	18
B	Standard Moore-Mealy machine for the same cell.	20

Cycle Accurate Simulation Model Generation for SoC Prototyping

Antoine Fraboulet
Citi, Insa-Lyon,
21 av. Jean Capelle
69621 Villeurbanne Cedex
antoine.fraboulet@insa-lyon.fr

Tanguy Risset, Antoine Scherrer
LIP, ENS Lyon,
46 allée d'Italie
69364 Lyon Cedex 07
tanguy.risset@ens-lyon.fr
antoine.scherrer@ens-lyon.fr

24th May 2004

Abstract

We present new results concerning the integration of high level designed IPs into a complete System on Chip. We first introduce a new computation model that can be used for cycle accurate simulation of register transfer level synthesised hardware. Then we provide simulation of a SoC integrating a data-flow IP synthesised with MMAAlpha and the SocLib cycle accurate simulation environment. This integration also validates an efficient generic interface mechanism for data-flow IPs.

Keywords system on chip, high level synthesis, VLSI, circuit simulation, computer aided design tools

1 Introduction

With the advent of multi-processors system on chip (MPSOC), fast cycle accurate hardware simulation is a major issue in the design process. Because most hardware design projects now include a significant software part, the simulation time increases a lot when it becomes precise enough to model highly unpredictable mechanisms such as cache misses or bus/Network contentions. In addition the ideal design scheme where one could easily re-use IPs designed elsewhere, as it is naturally done in software development, is far from today's habits mainly because of the lack of standardisation of communication protocols between IPs.

The SocLib initiative <http://soclib.lip6.fr/> proposes some advances in the resolution of these problems: IP standardisation and cycle accurate simulation time. First it proposes to write simulation models of IPs using SystemC at various level of refinement: *Transaction Level Modelling* (TLM) for software-like prototyping and *cycle accurate, bit accurate Modelling* (CABA) for a cycle accurate simulation of the IP. Then, it proposes to use VCI (Virtual Component Interface [1]) communication low level protocol to interface easily various IP. Finally it suggests a certain programming style that enables the possibility of writing very efficient simulation engines (closer to step by step simulation than to event driven simulation [2]).

During the design of a MPSOC, some IPs are re-used (processors, memories, DMA, etc.) but some (usually few of them) must be designed specifically for the application targeted. The

design of these new IPs and the integration in the global system must be fast to prototype rapidly the whole application. For many years now, people have been working on *high level synthesis* to reduce the time, effort and potential errors due to manual design of hardware. These research have lead to prototypes and industrial tools that perform this *semi-automatic* design of hardware from high level (functional or behavioural) specifications. One can check the following academic tools: MMAAlpha [3], Gaut [4], Paro [5], Compaan [6], Hugh [7] and the industrial initiative: Pico [8] (Synfora, spin off from HP), criticalBlue [9] and probably many more to come. Most of these tools address specific target circuits to reduce the search space during refinement, many of them target data-flow computations (sometimes referred to as *stream processing*) widely used in signal processing applications. This is the case for MMAAlpha that is used in this work to generate highly parallel hardware accelerators for signal processing filters. It is important to notice that the *de facto* failure of general high level synthesis tools such as *behavioural compiler* from Synopsys highlighted the fact that high level synthesis was intrinsically difficult to achieve and that the long research effort provided to produce the tools mentioned above was justified.

Most of these tools are now faced to the problem of integrating their resulting designs in MPSOC simulation and synthesis environments. The two most important problems encountered as soon as integration is envisaged are:

1. The choice of a communication protocol that provides data to the IP. This choice is driven by many factor: the simplicity of the protocol, the size and power of the hardware necessary for it and its impact on the performance of the global system.
2. The generation of a simulation model for the hardware synthesised which are compatible with the simulation platform used.

This paper explains which solutions were chosen for the MMAAlpha tool and it also extracts some underlying generic solutions that can be re-used for other tools. More precisely, we propose a new target computation model for the translation from classical structural RTL representation of hardware to finite state machine representation. We also present an efficient algorithm for performing the translation. This algorithm has been used for CABA simulation model generation from MMAAlpha. The generated simulation model have been integrated into a simple SOC composed of a MIPS R3000 processor, a standard RAM, a generic interconnect and a hardware accelerator generated with MMAAlpha. This platform runs a simple signal processing program performing filter on audio files. Apart from the validation of the above mentioned algorithms, this experiment was used to validate the SocLib approach and the facility of connecting external IPs generated by high level design tools in this framework.

Section 2 we introduce the new computation model that we would like to promote for high level design tools, some simulation performances are presented there. In Section 3 we present the soc simulation performed using SocLib and MMAAlpha. we have improve the interface mechanism between that hardware and software that was presented in [10] so that it can be reused for any data-flow IP in the SoC platform, simulation results concerning the whole SoC platform are presented in section 3.3.

2 FSM representation for RTL simulation models

Now that high level synthesis tools are gaining importance, the question is raised of how to generate efficient simulations models from internal representation of the tools which usually

corresponds to *classical* RTL representation. In this generation process, a crucial issue is the efficiency of the CABA simulation. The community now widely agree that finite state machine (FSM) is a good computation model for simulation because it can be easily understood by designers and it can be efficiently simulated. Following initial ideas of Jennings [11], many works [2, 12] are focussing on the problem of improving MPSOC simulation performances by reducing the inefficiency introduced by event driven simulation. In particular, the results of [2] state that if the FSM is represented in the form of Moore-Mealy machine, the simulation kernel can produce much more efficient simulations.

A moore-mealy machine is an automaton with an internal state and three functions: the `transition` function, the `moore` function and the `mealy` function. The standard definition of these functions is the following: the `transition` function uses the inputs and the current internal state to update the internal state. The `moore` function uses the internal state to provide outputs and the `mealy` function uses the inputs and the internal state to produce output (see the left of figure 3). In hardware component modelling, the internal state is composed of the internal register of the component and the state is computed at each clock cycle.

The idea experimented in [2, 12] is the following: if all the `mealy` functions of all the IPs are empty then there is no need for event driven simulation, a static scheduling can be performed and the simulation will run much faster. Of course, this is in general not the case but in most IPs the complexity of the `mealy` function will be negligible compared to the operations performed in the `moore` and `transition` function. Hence, if the simulation kernel is aware of this decomposition in the simulation models it can optimise the scheduling of the different functions to execute the `mealy` function as rarely as possible. In SystemCass [2] the way to write the `more`, `mealy` and `transition` functions is precisely specified and a static scheduling is built in the simulation kernel. In [12] the static scheduling is guided by the user via directives to the simulation kernel. Our approach does not need a particular scheduling strategy, we can use the standard SystemC simulation kernel. We improve SystemC simulation performances by suppressing redundant computations introduced by the standard Moore-Mealy automaton computation model. However we can further improve the performances by using the static scheduling strategies mentioned above.

2.1 A new FSM model for hardware modelling

Here, what we call a *classical* RTL representation is any language that specifies interconnection between boxes, these boxes being either registers or combinatorial operators (we assume the same common clock for each register). A nice graph modelling for this RTL representation is the one introduced by Jennings in [11] as *directed cyclic bipartite graph* (we call it the RTL graph in the following). In this representation, the components become one type of vertex and the nets become another type of vertex. We use the original Jennings' example in figure 1 to illustrate it, we denote by the *module* the whole hardware component to be abstracted. As mentioned by Jennings himself, we admit several hypothesis when building this representation: we know that the circuit will reach a stable state a each clock cycle (there is no combinatorial cycle), we know the direction of the flow of data (arcs are oriented), and we want to simulate *logical* behaviour rather than *physical* behaviour.

A straightforward algorithm for generating a Moore-Mealy machine from the RTL graph is the following: first identify the registers of the hardware (which will constitute the state of the machine). The input of these registers are computed by the `transition` function, for

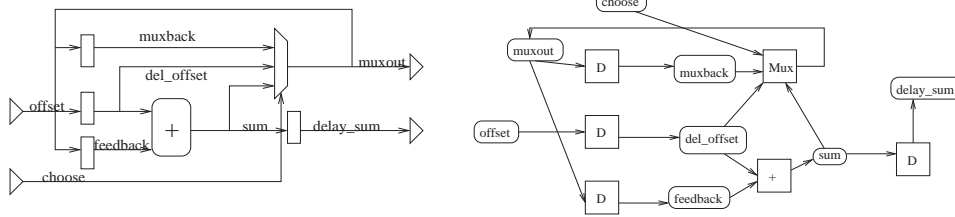


Figure 1: Original Jennings' RTL example and its representation as a RTL graph

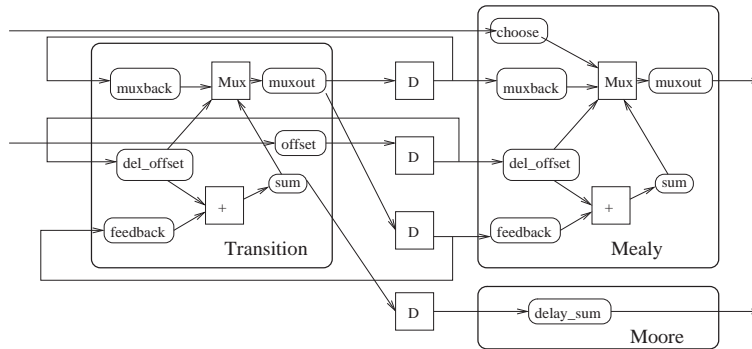


Figure 2: FSM representation of the hardware of figure 1 using a straightforward algorithm to build the moore, transition and mealy functions

each of these input, walk up the RTL graph and recursively add the computations encountered to the transition function until all paths have reached either a module input or the output of a register. To build the **moore** and **mealy** functions, start from the outputs of the module and recursively walk up the graph until all paths have reached either a module input or the output of a register. For a given output, if one of the path have reached a module input, the whole computation tree is placed in the **mealy** function otherwise it can be placed in the **moore** function. The application of this algorithm on the example of figure 1 gives the FSM of figure 2.

We believe that this straightforward algorithm has an important drawback that can be seen on figure 2: it duplicates code. Code duplication can be error prone (if the code is modified during the debugging process for instance) and it also implies an increased complexity during simulation because computation will be done twice. We propose to distribute the computations performed by the hardware in the three FSM functions without duplicating any one. For this we use a slightly modified definition of the three functions: the **moore** function inputs are unchanged, the **mealy** function will take as input some inputs of the module and some outputs of the **moore** function. Both functions **moore** and **mealy** will produce outputs that will be used by the transition function (while in the original definition, the **transition** function used only module inputs and register outputs). This is illustrated on the right of figure 3.

As long as FSM behaviour is respected, this new class of machine describes exactly the same set of machine as the standard Moore-Mealy. Indeed, from a machine of this new class, one can get a standard Moore-Mealy machine by duplicating some code of the functions. If this computation model is used for hardware simulation, we would like this new machine to behave exactly as the corresponding standard Moore-Mealy machine. This simply imply

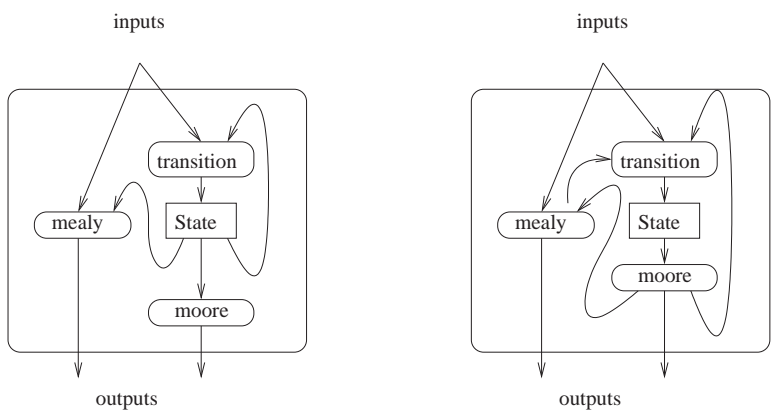


Figure 3: The standard Moore-Mealy machine used for writing CABA simulation models of IPs (on the left) and the extended definition that we used (on the right).

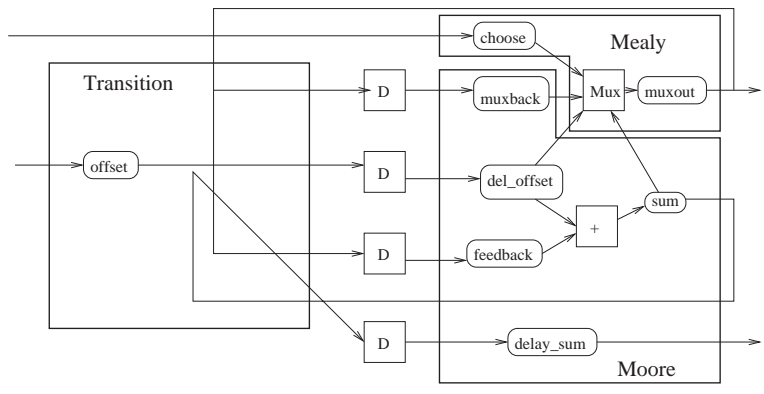


Figure 4: FSM representation of the hardware of figure 1 without duplicating code. The `transition` function is simply connecting nets.

that the simulation kernel must ensure the following property: the `mealy` function is evaluated at least one time before the `transition` function and at least one time after the `moore` function. We explain in the section 2.2 how to ensure that in SystemC.

With this target model in mind, a possible FSM for the module of figure 1 would be the one of figure 4 where no code duplication occurs. The construction of this FSM can simply consist of a rewriting of the RTL graph. Indeed, each hardware operator will be present in one and only one of the three functions: `transition`, `moore` or `mealy`. A clever traversal of the RTL graph will label each hardware operator of the graph with the type of function it belongs to (`transition`, `moore` or `mealy`). The nets will be classified too, but they may be duplicated if they are connected to operators of different type (we want the operands and results of an operator to be of the same type, this will ease the translation to a programming language).

The `labelRTLGraph` procedure described in figure 5 is a more precise description of the algorithm, in this description we have used the procedure `duplicate(type1,type2)` for the following action: split one net which is connected to two nodes of different types (namely `type1` and `type2`) into two nets linked with one directed arc, the first net will have type `type1`, the second net will have type `type2`. The inputs of the original net are input to the first net and the outputs of the original net are either transformed

```

labelRTLGraph()
  start with a RTL graph whith all nodes labelled unknown
  label all the register outputs as moore nets
  walk down the RTL graph from register outputs until arriving to outputs
    or to register inputs, and for each operator having all its operands
    labelled as moore:
    label the operator and its result as moore
  label all the unlabelled outputs as mealy nets
  walk up the RTL graph from these mealy outputs until arriving to inputs,
    or nets labelled moore and for each operator having its result
    labelled as mealy:
    label the operator and its operands as mealy
    if some of the operands are already labelled as moore:
      duplicate( moore, mealy)
  label all the remaining unknown nodes as transition
  if some register inputs are labelled with X, X≠transition:
    duplicate( X, transition)

```

Figure 5: Algorithm for labelling the RTL graph and generate a target FSM as the one presented in the right of figure 3

to output of the second net (if it reaches a node of type **type2**) or transformed to output of the first net (if it reaches a node of type **type1**). Note that we do not use the bipartite property of the graph and actually, the graph is not bipartite anymore after application of the **duplicate** procedure. For clarity we assumed that operators have a single result (but this can be easily extended to multiples results per operator). The algorithm assumes that it has a way of accessing inputs and outputs of the hardware (simply called *inputs* or *outputs* below), and to register inputs and outputs (called *register input* or *register output* below)

This algorithm first labels **moore** nodes with a forward traversal of the graph. These nodes are uniquely defined as they can only compute results from register outputs. Then it labels the **mealy** nodes starting from the outputs that are not **moore** with a backward traversal of the graph (each signal used in the **mealy** function must be **mealy** or **moore**). Finally the unlabelled signals and operators are necessary in the **transition** function. As we want the register inputs to come from the **transition** function, we duplicate the register inputs that are labelled **moore** or **mealy**. The RTL graph of figure 1 is represented on figure 6 after the execution of the algorithm, it corresponds exactly to the structuring exposed in figure 4 (the order in which nodes have been labelled is indicated in parenthesis).

This algorithm is *efficient* in the sense that its complexity is linear, assuming that one can have access in a constant time to registers. Indeed, each nodes of the graph is traversed only once (except for the duplications at the end of the algorithm but we can safely assume that the number of registers is small compared to the total number of nodes of the graph). It is *clever* in the sense that it does not duplicate any computation, hence the resulting simulation model of the hardware will be faster. The complexity of the graph of figure 2 and figure 4 illustrates it.

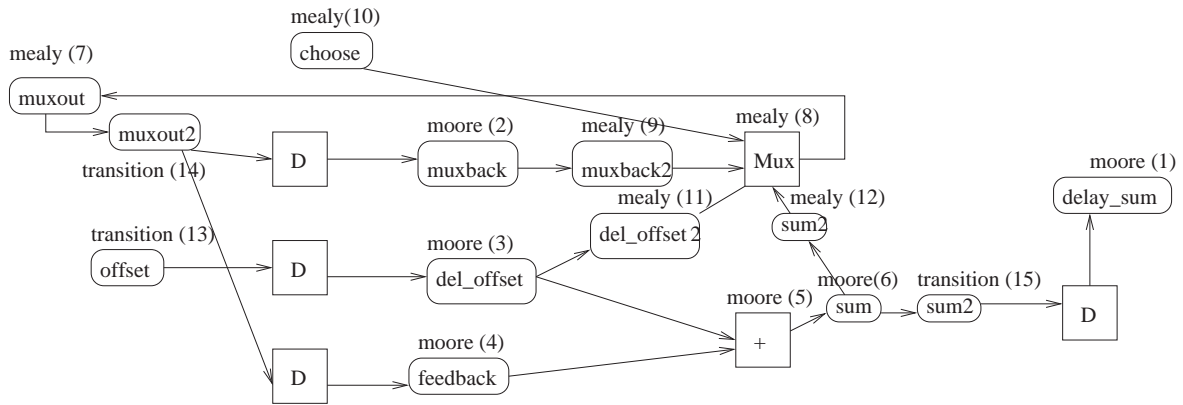


Figure 6: RTL graph of the hardware of figure 1 after the labelling algorithm presented here (the order in which nodes have been labelled is indicated). The resulting structuring of the FSM corresponds to the one of figure 4.

We have implemented this translation scheme in MMAAlpha to provide SystemC simulation model from MMAAlpha internal representation of RTL circuits. Simulation experiments are presented in the following section confirm a simulation time decrease of approximately 40%. The resulting simulation models have also been integrated in the platform presented in section 3.

2.2 SystemC implementation

In a SystemC implementation of this new FSM machine, we must ensure that the three methods corresponding to `mealy`, `moore` and `transition` functions are evaluated in a valid order. The usual way to ensure partial evaluation order in SystemC is to use the sensitivity list of the method in conjunction with the SystemC `sc_signal` type. For instance, if we use the classical FSM machine (left of figure 3), we can implement the registers of the module as `sc_signal`, set the `transition` method sensitive to the raise of the clock, set the `moore` method sensitive to the fall of the clock and set the `mealy` method sensitive to module inputs and register outputs. The semantic of the SystemC `sc_signal` type will ensure the correctness of the execution. More precisely, when the clock is low, the signal values will be the ones that occur on the real hardware.

In practice, If we consider a single module as the one of the left of figure 3 encapsulated into a test bench, the SystemC simulation kernel will use for its simulation the following order of evaluation of the methods: `mealy`, `transition`, `mealy`, `moore`. Of course, a real example connecting different modules together can result in much more complicated dynamic scheduling. As pointed in [2], the following order: `transition`, `moore`, `mealy` for a single module is also *always* valid with the classical FSM model. If now we use our new FSM (right of figure 3), this order is no more valid. We have to ensure the following conditions:

1. `mealy` must be executed at least once before `transition`,
2. `mealy` must be executed at least once after `moore`

Condition 2. above can be easily ensured by setting the `mealy` method sensitive to signals of the `moore` method it depends on. A more efficient scheme can use variables to communicate

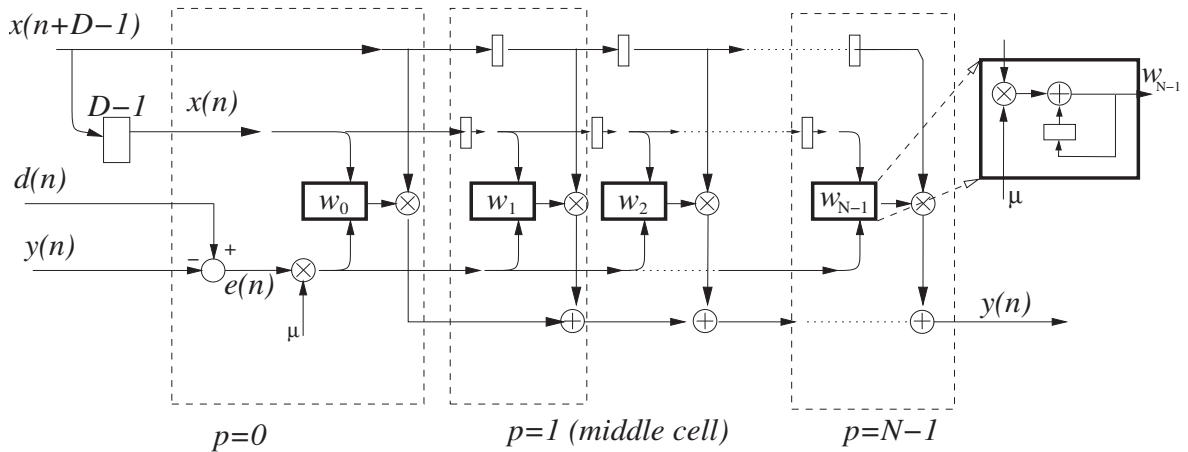


Figure 7: Non-pipelined DLMS architecture used for simulation performances. It contains a complete mealy part (computation of y), but also some moore computation (updates of the weight w).

between the methods instead of signals and set a dummy signal assigned at the end of the moore method that will trigger the execution of the mealy method. This is what we did in our implementation. Condition 1. cannot be ensured with the same trick because the transition method must be executed only once at each clock cycle, but this condition will naturally be enforced in any SystemC simulation kernel because the inputs of the module must be stable *before* the execution of the raise of the clock that will launch the execution of the transition method. Setting the mealy method sensitive to inputs it depends on will ensure condition 1. Finally the fact that the transition method uses results of the moore method rather than directly register outputs does not change anything. Hence a valid order for the new FSM for a single module will be: mealy, transition, moore, mealy. In practice, the SystemC simulation kernel will find this order.

For our experiments we have derived a specific hardware module: a non-pipelined version of the DLMS filter presented in [3]. The sketch of the DLMS architecture is presented on figure 7. The RTL version of the architecture has been derived with MMAAlpha from the same specification as the one used in [3]. The designer chose not to pipeline the architecture and hence obtained moore and mealy computations in each cell. Moreover, each mealy computations are connected from cell to cell, hence a dynamic scheduling more complex than the one mentioned above will take place. This architecture is composed of N cell with $N-2$ identical cells (we refer to this cell as middle cell, see figure 7). As explained in [3], we obtained a RTL representation of the circuit expressed in Alpha which can be translated syntactically into synthesizable VHDL. We have developed another translator that targets SystemC instead of VHDL. The translator first implement the algorithm presented in figure 5 to classify the nets of each cell and then use a syntax directed translation scheme to generate the SystemC code.

We generated SystemC code compliant with the model introduced above for this architecture with $N=30$ and $N=60$ cells. This gave rise to three types of cell: the first cell of the array, the last cell of the array and the middle cell repeated $N-2$ times. Each of these SystemC cell were implemented with this new FSM machine convention. Then we took the middle cell and modify it by hand so that it can stick to the classical Moore-Mealy machine. This consisted

	SystemC		SystemCass	
	new model	Moore-Mealy	new model	Moore-Mealy
Cell	0.58	0.76	0.13	0.2
module N=30	8.82	15.07	3.09	5.1
module N=60	17.56	30.63	6.14	10.34

Table 1: Simulation performances, in seconds, for a simulation of 100 000 clock cycles

in duplicating some of the code to use only cell inputs or register outputs in each method. It is worth noting that this represented an increase of 60% of the computations lines and 120% in the number of variables declared in the methods. The SystemC code of the new model cell automatically derived is presented in annex A and its corresponding Moore-Mealy cell is presented in annex B. Then we compared the SystemC simulation time for each cell (Moore-Mealy cell and new model cell) and for the whole array using either N-2 Moore-Mealy cell or N-2 new model cell. Finally we did the same comparison with the SystemCass simulation kernel to evaluate the improvement that would provide a static scheduling strategy.

These simulation were done with systemC-2.0.1 on linux operating system (processor pentium 4M, 1.4 GHz with 512 Mo ram). C++ Compilation was done with GCC 3.3 without any optimisation options. We also did the simulation with the SystemCass simulation kernel [2]. The results are presented in table 1.

These results show an improvement, between 30% and 50%, in the simulation performances obtained by SystemC simulation with the new model compared to equivalent hardware described as a classical Moore-Mealy machine. Simulation with SystemCass show that our improvement can be added to the simulation acceleration provided by static scheduling techniques. Note that simulation performance of SystemC and SystemCass should not be compared here because SystemCass currently does not implement real bit-true mechanism for handling the `sc_int` type (`int` type is used). The simulation time decrease is not huge, but we point out that this is not only the main advantage of our new coding style, it has other advantages: (i) it suppresses code duplication which is error prone, and (ii) it generates simulation models that are closer to final RTL implementation. This property could be used, for instance to derive automatically VHDL RTL models from the corresponding SystemC cycle accurate simulation models.

3 SoC platform simulation

This section present an experimental SoC simulation using SocLib and MMAAlpha using the results of the previous section. It first focus on an important problem: how to control easily hardware IPs that are designed by high level synthesis tool, then it presents the SoC platform simulated and some system and simulation performances.

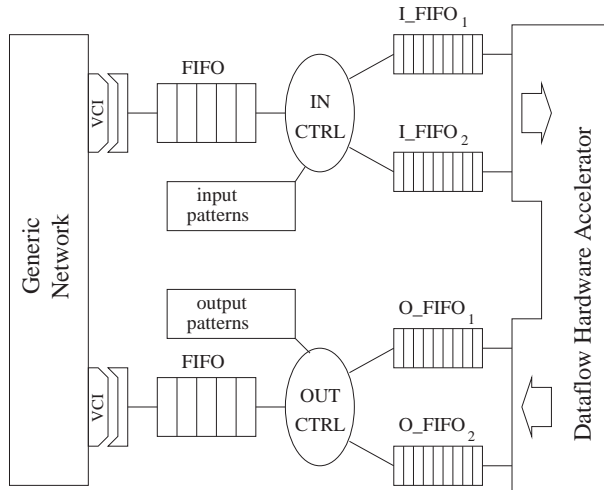
3.1 Generic interface for stream processing accelerators

We focus now on the problem of integrating data-flow hardware accelerators into a SoC platform. There are two important challenges in this topic. First, the communication protocol must be efficient because bandwidth is usually an important bottleneck for performances.

Second, the design time of this interface must be short, in particular for IPs synthesised by high level design tools. We present in this section a generic and parameterised hardware interface that can be used for simple or burst communications. This interface is an improvement over the one presented in [10] to integrate burst communication mode.

Our data flow hardware accelerator is abstracted as a black box with a pipelined behaviour, i.e. consuming and producing values at each clock cycle (see the "Data-flow hardware accelerator" in figure 8). As data may not be available due to timing issues, because the system cannot write values to the input port fast enough to feed the architecture for instance, we assume a clock enable mechanism that can freeze the execution of the architecture.

The left of figure 8 shows the hardware interface scheme for a pipeline with two inputs and two outputs streams. The right of figure 8 shows the *software interface*, or *driver*, running on a processor which controls the hardware accelerator. The generic hardware interface is composed of two main blocks, one for controlling the inputs and one for outputs, that are symmetrical. The controllers are used to dispatch incoming data to the correct input FIFO. The FIFOs between the VCI ports and controllers are used to hide latency that can occur on the interconnect (interruption occurring for instance). The LFIFOs and OFIFOs that are connected to the hardware accelerator are used to enable burst communications on the interconnect. To allow one clock cycle of the hardware accelerator, all the data read by the accelerator must be present in input FIFOs and all the FIFOs corresponding to output data must have empty slots. The hardware accelerator clock cycles are called *virtual clock cycle* as they do not necessarily correspond to the clock cycles of the system.



```
// input phase 0
for(i=0; i<16; i++)
{
    SEND(psl_mirr1,30);
    SEND(psr_mirr1,30);
}
// input phase 1
SEND(psl_mirr1,20);
SEND(psr_mirr1,20);
// output phase 0
RECV(pres_left,8);
RECV(pres_right,8);
// output phase 1
for(i=0; i < 16; i++)
{
    RECV(pres_left,30);
    RECV(pres_right,30);
}
// output phase 2
RECV(pres_left,18);
RECV(pres_right,18);
```

Figure 8: Hardware and Software interfaces for a Data-flow hardware accelerator. Controllers must be configured according the software communication pattern used in the software driver on the right hand side. Both the hardware and software can be parameterised to allow a maximum throughput using burst communications

As data-flow architectures can consume data of different bit-width synchronously, the data have to be grouped, according to the bit-width size of the interconnect to minimise the bandwidth needed for transfers. The order in which the data are transferred is very important to drive the architecture in an efficient way. The information of this order is stored into the interface input and output controllers in a set of configuration registers using a closed form encoding. We have introduced in [10] the concept of *Phase* and *Patterns* that allow this concise description, we briefly illustrate them with an example.

Figure 9 represents a data-flow hardware that uses three input data streams with different bit-width and the corresponding activation of each input stream at each clock cycle. A *phase* is a sequence of successive virtual clock ticks during which all inputs and outputs of the architecture are the same. A phase represent a regular behaviour of the interface. The example of figure 9 shows the definition of two phases: Φ_1 with only A input to the IP and Φ_2 where all variables are input. Inside each phases, possibly many different variables have to be sequentially sent into the FIFO. The *pattern* specify the order in which these data have to be send, figure 10 shows a valid pattern for phase Φ_2 of the IP of figure 9. This pattern is valid as soon as the size of the LFIFOs of figure 8 are at least one bus-width word in total size.

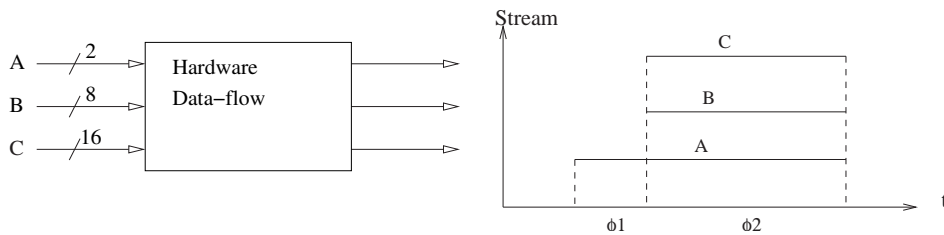


Figure 9: Hardware IP with three input streams and the corresponding input activation defining two input phases: Φ_1 and Φ_2

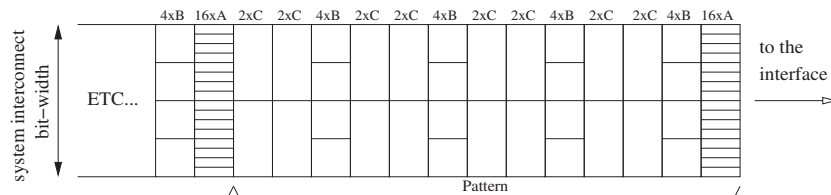


Figure 10: A valid pattern for the IP of figure 9

Using this concise information (phases and patterns) we can build communication schemes that are non-blocking and hence design the hardware and software interface. On the right of figure 8, one can see how the software driver uses the phases and the pattern inside each phase. Phases are used as superset of patterns to represent different configurations for sending data to the accelerator. They are generally used for computation initialisation and termination. For a more in-depth presentation of phases and patterns see [10] which introduced these concepts in hardware interface communications. These concepts are the key needed to provide interface and driver that can be re-used for other architecture possibly generated by other parties.

Burst communications allow to reduce the latency overhead between data arrivals. Using a different controller configuration for the architecture of figure 9 associated with the corresponding software driver, we can change the input patterns to the *burst pattern* presented

in figure 11. The main change is that several words of each data are grouped together for communication, this new pattern needs input L_FIFOs of size greater than one bus-width word: 8 for B for instance.

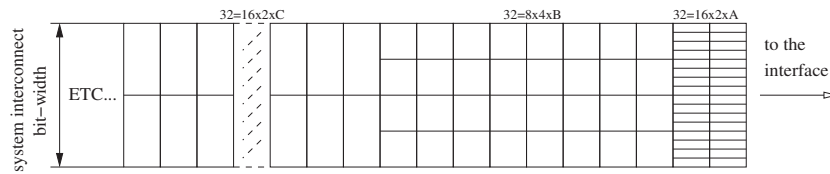


Figure 11: Bursted patterns for an architecture with three inputs of different bit-width (4, 8 and 16 bits)

For an IP generated with a high level synthesis tool, the phase and pattern can be either generated with the IP or written manually in a particular simple language. These information can then be used to generate the driver and to initialise the configuration registers of the interface. This is what we did with the IP generated by MMAAlpha as explained in the next section

3.2 Experimental platform setting

Choosing the good simulation platform is a crucial issue in MPSOC design because it implies a non-negligible investment for using the environment and developing the simulation models. Commercial tools may offer powerful environment but the lifetime of a tool will frequently be much shorter than the lifetime of an IP. People that developed some platform simulations with the VCC tool from Cadence have experimented that. We believe that the open source nature of SystemC (<http://www.systemc.org/>) is adapted to this kind of constraint. Nevertheless, we think that SystemC does not solve all the problems, in particular it does not sufficiently reduce simulation time for cycle accurate simulations. This is partly linked to the fact that SystemC simulation kernel heavily rely on event driven simulation.

3.2.1 SocLib and MMAAlpha

The main component of SocLib (<http://soclib.lip6.fr/>) is currently a set of SystemC simulation models for common IPs (MIPS processor, RAM, NoC, busses, DMA). These simulation models are publicly available and have been validated by extensive simulation at LIP6 laboratory. For each IP there exists a synthesizable RTL version of the IP that can be used for the final design (usually commercially available).

Anyone can simulate these IPs with the SystemC simulation kernel and build a complete MPSOC platform using SystemC simulation models coming from elsewhere as we did for the simulation models generated by MMAAlpha. The SocLib simulation models use the VCI (Virtual Component Interface [1]) communication low level protocol to interface with other IPs. It is also planned to normalise a higher level of modelling (*transaction level modelling*) for early prototyping stages.

In this framework, a MPSOC platform consists of a set of hardware IPs together with some software code running on the programmable IPs (here the MIPS processor), which imply the use of a compiler. We used GCC targeted to MIPS core. As we used a generic network on chip interconnect (and not a standard bus), it is mandatory to provide a simple operating

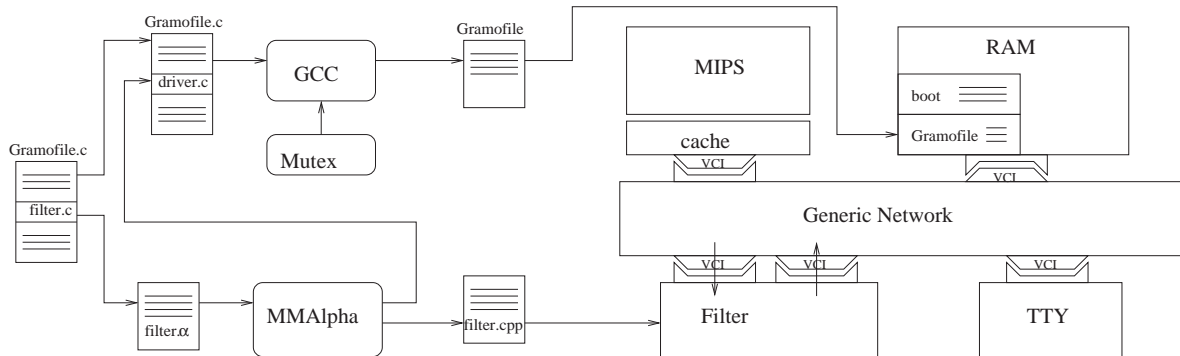


Figure 12: The SOC simulated and the global design methodology. Blank square box represent C++ simulation models of IPs.

system that ensure correct behaviour of the whole system (memory coherency, exception handling, etc.). Hence to be usable, SocLib must be associated to such an operating system compatible with the compiler used. We are currently using Mutek [13] which provides a very light implementation of Posix threads API.

MMAAlpha [3] is a toolbox for designing regular parallel architectures (systolic like) from recurrence equation specifications expressed in the Alpha language. It is one of the only existing tools that really automate the refinement of a software specification down to RTL description within the same language: Alpha. MMAAlpha's methodology for refinement successively introduces time (global synchronous clock), space (Mapping to 1 or 2-D array of processors), control generation and finally RTL level generation.

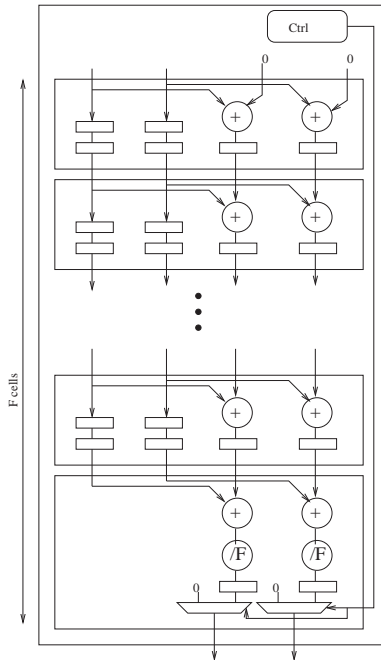
We have developed a translator from AlpHard (hardware description language, subset of Alpha) to SystemC, the underlying principles of this translator are explained in section 2. The translator also provides testbench that allow hierarchical debugging and which uses a stimuli file format which is common to the C code generator and the VHDL code generator of MMAAlpha.

3.2.2 The SOC platform

We have chosen a classical linux audio signal processing application called *Gramofile*. Gramofile processes audio files (.wav format) and proposes various simple filters like LP's tick removal. We have extracted a simple filter (referred as `filter.c`), and translated it in Alpha by hand. The translation was validated by replacing the original `filter.c` file by the C code generated from the `filter.alpha` by MMAAlpha. Through all the refinement stages of MMAAlpha, this C translation can be used to test the validity of the current description.

The target platform chosen is represented on figure 12, The hardware components of the platform are: a MIPS R3000 processor (with its associated data and instruction cache), a standard memory, a component used for displaying output (referred as TTY) and a specific hardware accelerator generated with MMAAlpha. All these components are connected via VCI ports to a simple network (internal architecture of this network is not precisely simulated, only the latency and bandwidth can be parameterised). The software running on the MIPS, in addition to bootstrapping information, is composed of the Gramofile program cross-compiled with GCC to a MIPS target.

In this paper we do not describe precisely the design of the accelerator with MMAAlpha



```

load["fsa.alpha"]; analyze[]; schedule[]
cGen["../gramo_rms_orig/fsa.c", {"F" -> 4},
    interactive-> False]
    (* uniformization *)
pipeall["acc_left", "sl.(n,i->n-i+F+1)",
    "Psl.(n,i->n+1,i+1)", "{n,i| i >= 0}" ];
pipeall["acc_right", "sr.(n,i->n-i+F+1)",
    "Psr.(n,i->n+1,i+1)", "{n,i| i >= 0}" ];
simplifySystem[]; ashow[]
    (* scheduling *)
schedule[scheduleType -> sameLinearPart,
    addConstraints -> {"TPslD1==1"}]
appSched[];
    (* RTL level derivation *)
toAlpha0v2[];
simplifySystem[alphaFormat->Alpha0];
convexizeAll[];
reuseCommonExpr[];
alpha0ToAlphard[];
    (* RTL VHDL and systemC generation *)
fixParameter["F", 4];
fixParameter["N", 1000];
a2v[];
a2sc[];

```

Figure 13: Hardware accelerator obtained with MMAAlpha for the example of a simple mean filter of F values and the commented MMAAlpha refinement script used to obtain it from high level specification.

(see [3] for details), the RTL representation obtained in Alphard for a simple mean filter is schematically represented on the left of figure 13 and the script that have been used in MMAAlpha to produce it from initial functional specification is presented on the right of figure 13 (MMAAlpha is programmed using the Mathematica software). One can see that the script is simple, the whole execution takes a few minutes. The architecture represented on the left of figure 13 is connected to the controller described in section 3.1 which connects on the bus and carefully dispatches the data coming from/to the memory on the right input/output ports. This driver is also generated by MMAAlpha together with a set of parameters that tunes the controller accordingly.

3.3 System and simulation performances

The SoC simulation results are presented in the table 2. In this implementation we used a simple memory transfer mechanism driven by the processor. This dual read/write generates a significant overhead due to memory access and system interconnect latency taken into account for each word transferred by the processor to the accelerator or back to memories. This overhead can be overcome by using a DMA engine to speedup memory copies using burst transfers between the memories and the hardware accelerator.

complete simulation time	29.21 s
complete simulation cycles	600000 cycles
simulation speed	20540.9 cycles/seconds
hardware pipeline throughput	20450 cycles

Table 2: Performances obtained from the complete SoC simulation

It is interesting to compare these results with approach of Quinton et al. [14] where a very similar application is prototyped by using emulation on a real platform (Lyrtech SignalMaster, with a DSP and a FPGA) rather than simulation. Their implementation uses an API provided by Lyrtech for hardware-software interface (roughly the equivalent of our interface controller and driver), unfortunately we have no precise estimation of the performance of the interface. In [14] the simulation time of the complete application per filter sample is approximately $6\mu s$ while our simulation reaches $29.21/20450 \simeq 1.4ms$ per filter sample (250 times slower for a simpler application). Hence, the prototyping approach is much better as soon as simulation time is targeted. However, if cycle accurate simulation is sought, our interface mechanism is much closer to what will be on the final chip than the one used in [14] and the behaviour of our simulation is exactly what will occur on the real system.

4 Conclusion

In this report we have presented an efficient computation model for performing cycle accurate hardware simulation of RTL description of hardware. This representation is particularly useful for high level synthesis tools that provides *generated* simulation models rather than manually written simulation models. We have validated the simulation time improvements provided by this new computation model but we believe that the quality of the code generated is a more important advantage of our model.

We have also presented a complete SoC simulation integrating IPs coming from different places: manually written IPs developed for standard SoC platform and a hardware accelerator IP generated with MMAAlpha and associated with a generic hardware interface and software driver, both generated by MMAAlpha. This integration is a case for the use of the SocLib environment and it also highlighted the good efficiency of our generic hardware/software mechanism.

References

- [1] Alliance, V.: Virtual component interface standard (ocb specification 2, version 1.0) (2000)
- [2] Pétrot, F., Hommais, D., Greiner, A.: A simulation environment for core based embedded systems. In: Annual Simulation Symposium, Atlanta, GA, U.S.A (1997) 86–91
- [3] Guillou, A.C., Quinton, P., Risset, T., Wagner, C., Massicotte, D.: High level design of digital filters in mobile communications. Technical Report 1405, Irisa (2001)

- [4] Sentieys, O., Diguët, J., Philippe, J.: Gaut: a high level synthesis tool dedicated to real time signal processing application. In: EURO-DAC. (2000) University booth stand.
- [5] Bednara, M., Teich, J.: Interface synthesis for fpga based vlsi processor arrays. In: Proc. of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA 02), Las Vegas, Nevada, U.S.A. (2002)
- [6] Kienhuis, B., Rijpkema, E., Deprettere, E.: Compaan: Deriving process networks from matlab for embedded signal processing architectures. In: 8th International Workshop on Hardware/Software Codesign (CODES'2000). (2000)
- [7] Augé, I., Donnet, F., Gomez, P., Hommais, D., Pétrot, F.: Disydent: a pragmatic approach to the design of embedded systems. In: Design, Automation and Test in Europe Conference and Exhibition (DATE'03 Designers' Forum), Paris, France (2002)
- [8] et al., R.S.: High-Level Synthesis of Non Programmable Hardware Accelerators. In: IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP 2000), Boston (2000)
- [9] Hounsell, B., Taylor, R.: Co-processor synthesis a new methodology for embedded software acceleration. In: Design, Automation and Test in Europe Conference and Exhibition (DATE'03 Designers' Forum), Paris, France (2004)
- [10] Derrien, S., Guillou, A.C., Quinton, P., Risset, T., Wagner, C.: Automatic synthesis of efficient interfaces for compiled regular. In: International Samos Workshop on Systems, Architectures, Modeling and Simulation (Samos), Samos, Grece (2002)
- [11] Jennings, G.: A case against event-driven simulation for digital system design. In: Proceedings of the 24th annual symposium on Simulation, IEEE Computer Society Press (1991) 170–176
- [12] Pérez, D.G., Mouchard, G., Temam, O.: A new optimized implementation of the systemc engine using acyclic scheduling. In: Design, Automation and Test in Europe Conference and Exhibition (DATE'03 Designers' Forum), Paris, France (2004)
- [13] Pétrot, F., Gomez, P.: Lightweight implementation of the posix threads api for an on-chip mips multiprocessor with vci interconnect. In: Design, Automation and Test in Europe Conference and Exhibition (DATE'03 Designers' Forum), Munich, Germany (2003)
- [14] Charot, F., Nyamsi, M., Quinton, P., Wagner, C.: Architecture Exploration for 3G Telephony Applications Using a Hardware–Software Prototyping Platform. In: Proc. of Samos IV, Samos, Greece (2004)

A New FSM model for one cell of the DLMS

```
// SystemC Model Created for "system cellfirrModule3"
//-- 2/4/2004 10:51:22
// Alpha2SystemC

/*cellfirrModule3.h*/

#include <systemc.h>

struct cellfirrModule3 : sc_module{
  /*ports*/
  sc_in<clk>          clock;
  sc_in<bool>         reset;
  sc_in<sc_int<16>> > X_reg3_loc;
  sc_in<sc_int<16>> > XD_reg2_loc;
  sc_in<sc_int<16>> > Y_reg7;
  sc_in<sc_int<16>> > ED_reg1;
  sc_in<bool> > W_ctl1_In;
  sc_out<sc_int<16>> > X;
  sc_out<sc_int<16>> > XD;
  sc_out<sc_int<16>> > Y;
  sc_out<sc_int<16>> > ED;

  int procNum;// processor number

  /* registers: internal state of the cell: updated with
   the transition method, */
  sc_signal<sc_int<16>> > EDloc4TR;
  sc_signal<sc_int<16>> > WTR;
  sc_signal<sc_int<16>> > XD_reg2_loc_IO;
  sc_signal<sc_int<16>> > XDloc2TR;
  sc_signal<sc_int<16>> > X_reg3_loc_IO;
  sc_signal<sc_int<16>> > Xloc1TR;

  sc_signal<bool> > trigger_mealy;

  /*variables of the Moore method used in other method */
  sc_int<16> TSep3;
  sc_int<16> XDloc2;
  sc_int<16> Xloc1;
  sc_int<16> X_reg8;

  /*variables of the Mealy method used in other method */
  sc_int<16> EDloc4;
  sc_int<16> W;
  sc_int<16> TSep3M;
  sc_int<16> X_reg8M;

  void transition();

  void moore();

  void mealy();

  ////////////////////////////////////////////////////////////////////
  // constructor
  ////////////////////////////////////////////////////////////////////
  SC_HAS_PROCESS(cellfirrModule3);
  cellfirrModule3(sc_module_name insname,int p ){

    SC_METHOD(transition);
    sensitive_pos << clock;

    SC_METHOD(moore);
    sensitive_neg << clock;

    SC_METHOD(mealy);
    sensitive << reset;
    sensitive << X_reg3_loc;
    sensitive << XD_reg2_loc;
    sensitive << Y_reg7;
    sensitive << ED_reg1;
    sensitive << W_ctl1_In;
    sensitive << trigger_mealy;

    procNum = p;
    trigger_mealy=1;
  }
};

// SystemC Model Created for "system cellfirrModule3"
//-- 2/4/2004 10:51:22
// Alpha2SystemC

/*cellfirrModule3.cpp*/
```

```

void cellfirrModule3::transition(){

    /*local variables of the Transition method */

    /*equations the transition Method */

    XD_reg2_loc_IO = XD_reg2_loc;
    X_reg3_loc_IO = X_reg3_loc;
    EDloc4TR = EDloc4;
    XDloc2TR = XDloc2;
    Xloc1TR = Xloc1;
    WTR = W;
#ifdef DEBUG
    printf("tr %d",procNum);
#endif
} // end process

void cellfirrModule3::moore(){

    /*local variables of the Moore Method */
    sc_int<16>    ED_reg5;
    sc_int<16>    TSep1;
    sc_int<16>    W_reg4;
    sc_int<16>    XD_reg2;
    sc_int<16>    XD_reg6;
    sc_int<16>    X_reg3;

    /*equations the Moore Method */

    ED_reg5 = EDloc4TR;
    W_reg4 = WTR;
    XD_reg2 = XD_reg2_loc_IO;
    XD_reg6 = XDloc2TR;
    X_reg3 = X_reg3_loc_IO;
    X_reg8 = Xloc1TR;
    XDloc2 = XD_reg2;
    TSep1 = (ED_reg5 * XD_reg6);
    Xloc1 = X_reg3;
    XD.write( (sc_int<16> )XDloc2);
    TSep3 = (W_reg4 + (TSep1 / 32768));
    X.write( (sc_int<16> )Xloc1);
    //trigger the execution of mealy
    trigger_mealy = !((bool)trigger_mealy);
#ifdef DEBUG
    printf("mo %d",procNum);
#endif
} // end process

void cellfirrModule3::mealy(){

    /*local variables of the Mealy Method */

    /*equations the Mealy Method */

    EDloc4 = ED_reg1;
    X_reg8M = X_reg8;
    ED.write( (sc_int<16> )EDloc4);
    TSep3M = TSep3;
    W = (W_ctl1_In ? ((sc_int<16> )0) : (TSep3M));
    Y.write( (sc_int<16> )(Y_reg7.read() + ((W * X_reg8M) / 32768)));
#ifdef DEBUG
    printf("mi %d",procNum);
#endif
} // end process

```

B Standard Moore-Mealy machine for the same cell.

```
// SystemC Model Created for "system cellfirrModule3"
//-- 2/4/2004 10:51:22
// Alpha2SystemC

/*cellfirrModule3.h*/

#include <systemc.h>

struct cellfirrModule3 : sc_moduleif
/*ports*/
sc_in<clk> clock;
sc_in<bool> reset;
sc_in<sc_int<16>> X_reg3_loc;
sc_in<sc_int<16>> XD_reg2_loc;
sc_in<sc_int<16>> Y_reg7;
sc_in<sc_int<16>> ED_reg1;
sc_in<bool> W_ct11_In;
sc_out<sc_int<16>> X;
sc_out<sc_int<16>> XD;
sc_out<sc_int<16>> Y;
sc_out<sc_int<16>> ED;

int procNum;// processor number

/* registers: internal state of the cell: updated with
the transition method, */
sc_signal<sc_int<16>> EDloc4TR;
sc_signal<sc_int<16>> WTR;
sc_signal<sc_int<16>> XD_reg2_loc_I0;
sc_signal<sc_int<16>> XDloc2TR;
sc_signal<sc_int<16>> X_reg3_loc_I0;
sc_signal<sc_int<16>> Xloc1TR;

/*variables of the Moore method used in other method */
/*variables of the Mealy method used in other method */

void transition();

void moore();

void mealy();

//////////
// constructor
//////////
SC_HAS_PROCESS(cellfirrModule3);
cellfirrModule3(sc_module_name insname,int p ){

    SC_METHOD(transition);
    sensitive_pos << clock;

    SC_METHOD(moore);
    sensitive_neg << clock;

    SC_METHOD(mealy);
    sensitive << reset;
    sensitive << X_reg3_loc;
    sensitive << XD_reg2_loc;
    sensitive << Y_reg7;
    sensitive << ED_reg1;
    sensitive << W_ct11_In;
    sensitive << EDloc4TR;
    sensitive << WTR;
    sensitive << XD_reg2_loc_I0;
    sensitive << XDloc2TR;
    sensitive << X_reg3_loc_I0;
    sensitive << Xloc1TR;

    procNum = p;
}
};

// SystemC Model Created for "system cellfirrModule3"
//-- 2/4/2004 10:51:22
// Alpha2SystemC

/*cellfirrModule3.cpp*/

void cellfirrModule3::transition(){

/*local variables of the Transition method */
sc_int<16> EDloc4;
sc_int<16> W;
```

```

sc_int<16>   TSep3M;
sc_int<16>   TSep3;
sc_int<16>   TSep1;
sc_int<16>   Xloc1;
sc_int<16>   X_reg3;
sc_int<16>   XDloc2;
sc_int<16>   XD_reg2;
sc_int<16>   X_reg8M;
sc_int<16>   X_reg8;
sc_int<16>   XD_reg6;
sc_int<16>   ED_reg5;
sc_int<16>   W_reg4 ;

/*equations the transition Method */

XD_reg2_loc_I0 = XD_reg2_loc; //OK, input
X_reg3_loc_I0 = X_reg3_loc;  //OK, input
EDloc4 = ED_reg1;           //input, OK
EDloc4TR = EDloc4;         //OK
XD_reg2 = XD_reg2_loc_I0;   //added, input
XDloc2 = XD_reg2;          // added
XDloc2TR = XDloc2;         //OK
X_reg3 = X_reg3_loc_I0;    //added, reg
Xloc1 = X_reg3;            //added
Xloc1TR = Xloc1;           //OK
XD_reg6 = XDloc2TR;       //added, reg
ED_reg5 = EDloc4TR;       //added, reg
TSep1 = (ED_reg5 * XD_reg6); //added
W_reg4 = WTR;              //added, reg
TSep3 = (W_reg4 + (TSep1 / 32768)); //added
TSep3M = TSep3;           //added
W = (W_ctl1_In ? ((sc_int<16> )0) : (TSep3M)); //added
WTR = W;
#ifdef DEBUG
    printf("tr %d",procNum);
#endif
} // end process

void cellfirrModule3::moore(){

/*local variables of the Moore Method */
sc_int<16>   ED_reg5;
sc_int<16>   TSep1;
sc_int<16>   W_reg4;
sc_int<16>   XD_reg2;
sc_int<16>   XD_reg6;
sc_int<16>   X_reg3;
sc_int<16>   TSep3;
sc_int<16>   XDloc2;
sc_int<16>   Xloc1;
sc_int<16>   X_reg8;

/*equations the Moore Method */

ED_reg5 = EDloc4TR;        //OK
W_reg4 = WTR;              //OK
XD_reg2 = XD_reg2_loc_I0;  //OK
XD_reg6 = XDloc2TR;       //OK
X_reg3 = X_reg3_loc_I0;    //OK
X_reg8 = Xloc1TR;         //OK
XDloc2 = XD_reg2;         //OK
TSep1 = (ED_reg5 * XD_reg6); //OK
Xloc1 = X_reg3;           //OK
XD.write( (sc_int<16> )XDloc2); //OK
TSep3 = (W_reg4 + (TSep1 / 32768)); //OK
X.write( (sc_int<16> )Xloc1); //OK
#ifdef DEBUG
    printf("mo %d",procNum);
#endif
} // end process

void cellfirrModule3::mealy(){

/*local variables of the Mealy Method */
sc_int<16>   EDloc4;
sc_int<16>   W;
sc_int<16>   TSep3M;
sc_int<16>   X_reg8M;
sc_int<16>   X_reg8;
sc_int<16>   XD_reg6;
sc_int<16>   ED_reg5;
sc_int<16>   TSep1 ;
sc_int<16>   TSep3 ;
sc_int<16>   W_reg4 ;

/*equations the Mealy Method */

EDloc4 = ED_reg1;          //input, OK

```

```
X_reg8 = Xloc1TR;          //added, reg
X_reg8M = X_reg8;         // OK
ED.write( (sc_int<16> )EDloc4); //OK
XD_reg6 = XDloc2TR;      //added, reg
ED_reg5 = EDloc4TR;     //added, reg
TSep1 = (ED_reg5 * XD_reg6); //added
W_reg4 = WTR;           //added, reg
TSep3 = (W_reg4 + (TSep1 / 32768)); //added
TSep3M = TSep3;        //OK
W = (W_ctl1_In ? ((sc_int<16> )0) : (TSep3M)); //OK
Y.write( (sc_int<16> )(Y_reg7.read() + ((W * X_reg8M) / 32768))); //OK
#ifdef DEBUG
    printf("mi %d",procNum);
#endif
} // end process
```