High Level Analysis of Clock Regions in a C++ System Description

Luc Rynders
DESICS/DBATE
IMEC
B-3001 Leuven
rijnders@imec.be

Patrick Schaumont
DESICS/DBATE
IMEC
B-3001 Leuven
schaumont@imec.be

Serge Vernalde
DESICS/DBATE
IMEC
B-3001 Leuven
vernalde@imec.be

Ivo Bolsens
DESICS
IMEC
B-3001 Leuven
bolsens@imec.be

Abstract— Timing verification of digital synchronous designs is a complex process that is traditionally carried out deep in the design cycle, at the gate level. A method, embodied in a C++ based design system, is presented that allows modeling and verification of clock regions at a much higher level. By combining event-driven, clock-cycle true and behavioral simulation, we are able to perform static and dynamic timing analysis of the clock regions. In addition a significant increase in the design cycle speed is obtained.

I. Introduction

Let's first consider a small example, that illustrates the targeted applications. In a cable communication system, there is a need for a flexible setting of the carrier frequency. At the source side the system contains a transmitter, which converts the digital input data into a complex QAM modulated signal. This band-limited QAM signal is up-converted to a selectable carrier frequency before it is send over the cable. At the receiver side the band of interest is down-converted and demodulated to retrieve the digital data.

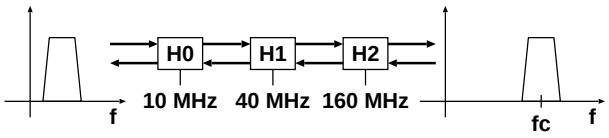

Fig. 1. Architecture of flexible digital up and down conversion requires communicating clock regions.

The architecture of the digital filter banks, shown in Fig. 1, allows for flexible setting of the carrier frequency both for up- and down-conversion [9]. It consists of programmable multi-rate filters and complex rotators. It is divided in 3 parts that are clocked at different frequencies (160, 40 and 10 MHz), where the lower frequency clocks are derived from the highest frequency clock.

From the specified behavior it is clear that the hardware in the 3 clock regions need to communicate data in both directions : for up- and for down-conversion.

Design experience shows that designing the separate filter stages is fairly easy compared to all interconnection problems that are expected between the different clock regions internally, as well as with the external components. More over, the timing problems of these interconnections only start to appear when the design is presented at the gate level and all detailed timing effects are taken into account [1].

We have developed a method in a C++ based design environment that allows us to analyze this problem effectively at a higher level. The design environment [8] supports the design of hardware circuits by means of an object library. It has an elaborated code generation back end that produces synthesizable HDL code and test benches from the high level C++ description.

Our first concern is thus that we want to describe the design including communication aspects over the clock regions. The design approach we follow is discussed in Section 2. In Section 3 we present the computational model [4] behind the design description. With a classical design approach, the clock system is only checked and verified at the gate level [1], especially regarding skew, minimum width, setup/hold time, critical path). In Section 4 we show that similar checks are possible at the RT-level within the high-level design environment.

The last sections show the path to implementation and summarize the results. We also highlight the benefits of the design strategy used.

II. Design Approach

Our design approach is based on a flexible, extendible design environment in C++, making use of an object library [5]. This allows the designer to use the same environment for executable specification and test bench description down to the RT-level architecture. The environment further allows to use the same description for simulation, for hardware code generation and for lower level test bench generation. During the description of the architecture and the simulation, numerous verifications and analysis are possible. We first concentrate on the architectural description related to the clock regions and in the following sections, the simulation and verification issues

are highlighted.

The design is described by using an object hierarchy for synchronous digital circuits. The basic building blocks of the architecture are finite state machines with data paths (FSMDs). These blocks are interconnected via communication channels. The data paths consist of signal flow graphs (SFGs), while the FSMs contain states and transitions between these states. At each transition, SFGs are selected to be executed.

Other types of blocks e.g. to describe test benches or memories can be integrated into the same description as the designer has all programming language features at his disposal [8]. Whenever the design at-hand uses new design concepts, the object library can be extended to include them. This is what is done for the application described above by using the concepts that are related to clock signals and clock regions. New classes of objects and functions are defined for these concepts to incorporate the presented timing verification.

Current approaches to timing verification are defined either at the system level ([2, 3]) or else at the structural gate level. Our contribution is to define a timing verification framework at the RT-level. This allows structural timing verification without doing costly logic synthesis iterations.

A. Design Example

To illustrate the architectural description of the FSMDs and the clock generation concepts, an example of a clock divider by a factor of 4 (from 160 MHz to 40 MHz) is given below :

```
//-- Define 160 MHz clock
CLK_FIX (ck_160, 6);
CLK_INV (ck_160_inv, ck_160);

//-- Clock divider
FSMD (clock_div4, ck_160_inv);

  //- Data Path
  REG (ck_gen);
  SFG (clk0); ck_gen = 0;
  SFG (clk1); ck_gen = 1;

  //- Finite State Machine
  STATE (s0); STATE (s1);
  STATE (s2); STATE (s3);
  AT (s0) DO (clk0) GO (s1);
  AT (s1) DO (clk0) GO (s2);
  AT (s2) DO (clk1) GO (s3);
  AT (s3) DO (clk1) GO (s0);

//-- Promote data signal to clock
CLK_REG  (ck_40, ck_gen);
```

We first define the independent clock `ck_160` of 160 MHz, with a clock period of 6 ns. A simple inversion is enough to generate the inverted clock `ck_160_inv`. This inversion is however not described as such but as a clock object `CLK_INV`. In this way the semantics of a clock signal are valid for the inverted clock.

The next part describes the FSMD, called `clock_div4`, which will enable us to form the lower rate clock signal. The FSMD is clocked by `ck_160_inv`. It contains 1 register `ck_gen`, which is also an explicit object. All registers inside the FSMD are clocked at the clock of the FSMD. The data path contains 2 SFGs, which set the register to a fixed value (0 or 1). The controller contains 4 states, which are traversed in a loop. This generates a data waveform 0,0,1,1,... at the register output.

This register output is still a data signal. It is semantically translated into a clock signal by the clock object `CLK_REG`, called `ck_40`, that can drive a different clock region. The clock divider and its clock waveforms are shown in Fig. 2.

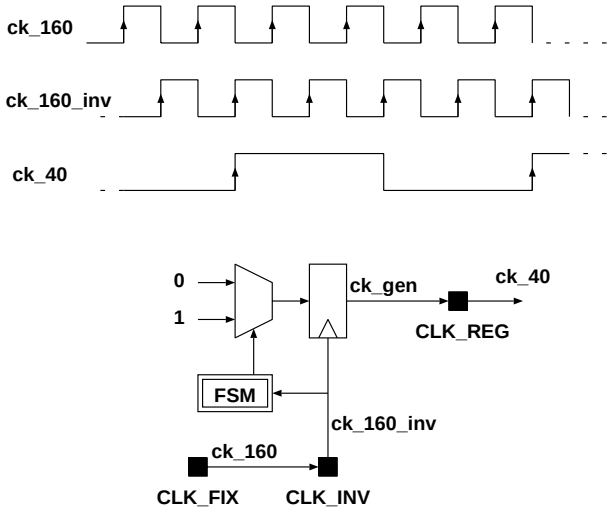

Fig. 2. Correct clock waveforms are generated by a clock divider containing an FSMD and 3 clock objects.

III. Simulation strategy

The simulation strategy in the C++ design environment is based on the usage of quantum of computation (QoC), which means that all processing is done in small pieces. Whenever a block is allowed to run, it will perform some little action and then give control back. The control which triggers the execution of the computation quanta is external to the blocks.

This strategy implies that there is no strictly imposed master simulation model, and it allows that different design entities can be used within a higher level simulator. It also allows to realize co-simulations with other blocks or descriptions styles, e.g. for hardware-software co-design or instruction-level simulations [7].

A. Block behavior

Data Flow : When control is given to a data flow block, it first checks by means of a firing rule if enough

tokens are present at its input queues. In this way it is possible to adhere to the data flow semantics. After the processing or when the firing rule is not satisfied, the data flow block returns control to the caller, and the QoC for this block is finished.

FSMD : When a FSMD is triggered, it selects a transition to a next state based on the current state and the input condition values. The SFGs, associated to the selected transition become active. Data path operations can read data from input channels and write data to the output channels. After the execution of these action, which all happen in 1 clock period, control is returned and the QoC is finished.

Clock : When a clock block is given control it determines the new value of the clock signal. This is a simple boolean operation (for a derived clock) or a function of time (for an independent clock).

B. Communication

Data flow queues : A first communication channel is through data flow queues (FIFOs). They are meant to interconnect the data flow blocks, but they can as well be used between FSMDs.

Data value : A second type of communication is a direct connection. At the input side a data token can be put on it and the data value remains valid until the next token is put, overwriting the current value.

It has been shown that this type of communication is useful to combine data flow system level with event driven simulations [6].

Clock : Clock signals are also seen as communication channels. Whenever an event occurs at a clock signal it is communicated to all blocks that are in some way related to it : derived clocks and FSMDs that trigger on the active clock edge.

C. Hierarchy of Computation

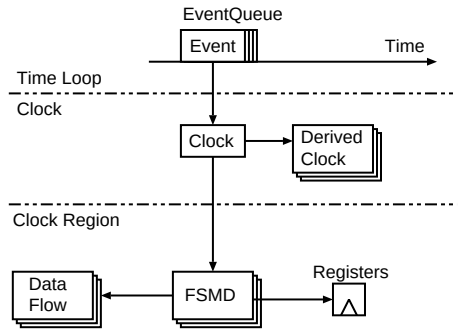

Fig. 3. A hierarchical computation model combines event-driven, clock-cycle true and data flow simulation.

A hierarchical computation model, putting all the blocks and communication channels together, is constructed in the following way (see Fig. 3) :

Time loop : At the highest level of computation, we use a loop over the simulation time until a specified end time. Time is advanced when nothing remains to be processed at the current point in time. To be able to know what is to be processed, a global queue of events is used. This simulation engine is summarized in the following code :

```
// main time loop
while (curTime < endTime) {
  // delta time simulation
  while (eventQ.getTime() == curTime) {
    // process current event
    curEvent = eventQ.pop();
    curEvent.run();
  }
  // advance time to next event
  curTime = eventQ.getTime();
}
```

Clock event : The event queue contains only the events of the clock signals. Because clocks can be dependent on each other, all events on clock signals must be propagated. Propagation is done by running each of the clock blocks that can be activated by the current clock event. New events are added to the queue at the same time point (delta time) or at some future time point for independent clocks. Whenever a clock event has an active transition, the clock region is activated with its run() method. This simulation is summarized in the following code :

```
// current clock
cur_clock = curEvent.clock();
cur_clock.run();
// derived clocks
FOREACH (derived_clock, cur_clock) {
  derived_clock.run();
}
// clock region
if (cur_clock.active_edge()) {
  cur_region = cur_clock.region();
  cur_region.run();
}
```

Clock region : On activation of a clock region, each of the FSMDs of the clock region is activated. Whenever a token is put on some data flow queue, the data flow block at the receive side of the queue is also given control to run. This is needed because data flow blocks do not belong to a specific clock region and also need activation from the simulation engine. When all computations as belonging to the current clock cycle are done, the registers are updated. This simulation behavior is summarized in following code :

```
// FSMD blocks
FOREACH (fsmd, cur_region) {
  fsmd.run();
  // Data Flow Blocks
  FOREACH (output_queue, fsmd) {
    if (output_queue.new_token()) {
      output_queue.receiver().run();
} } }
```

```
// Update registers
FOREACH (fsmd, cur_region) {
  fsmd.registers().update();
}
```

IV. Verification and Analysis

Using the concepts of clocks and clock regions, specific verification and analysis methods related to these classes become possible.

In order to verify successfully timing properties at the RT level, we define a minimum set of clock attributes. The following properties are essential features of the clock signal at the RT level : RT minimum width, RT setup and RT hold time. These properties are defined using the `CLK_TIME` method on the clock object. An example is shown below for the `ck_40` clock and is illustrated in Fig. 4 : `CLK_TIME (ck_40, 12, 9, 9);`

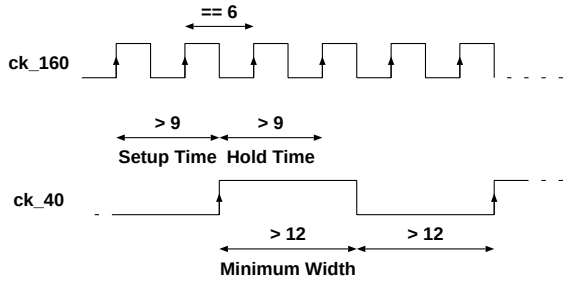

Fig. 4. The timing constraints for the ck_40 region span several clock cycles of the ck_160 region. They are automatically verified during simulation.

Most requirements of proper clock generation and clock relations between different clock regions can not be analyzed statically. Indeed, the generation of the clock edges can be a complicated combination of other clock signals and might also be data dependent or controlled by the states of a FSM. The following checks are performed and are illustrated in Fig. 5.

RT Minimum Width : For a proper operation of the clock regions, we require that the clocks satisfy the minimum width requirement. At the gate level this minimum width specifies the minimum time a clock level must be stable to enable the proper operation of a register element (flip-flop). This type of minimum width is identical for each clock as it is not related to the clock region but to a register cell.

At the RT-level, this requirement is reformulated to be the minimum time that the clock signal must be stable for proper functioning of the whole clock region. Using a symmetric clock waveform, this minimum width is half the clock period. As the clock period might be different for each clock region, so may be the minimum width.

RT Setup and Hold Time : For proper operation of the flip-flop not only the clock signal must be sufficiently stable, but also the data signal must be stable at the

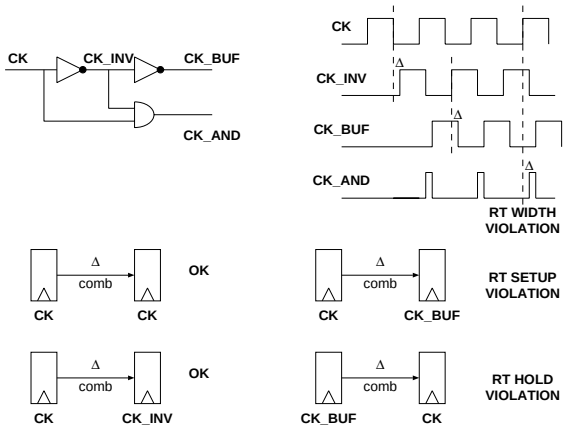

Fig. 5. Overview of the dynamical checks of clock timing rules that are verified during simulation.

active edge of the clock. This means that the data must be stable some time before (i.e. setup time) and after (i.e. hold time) the clock edge.

At the RT-level we do not know the exact delays of the data signals, except that they are constrained to fit within the clock period. We assume that these delays take minimum time, which means that the SFGs are simulated in delta time in the same way as is done for the event driven clock signals. In this way possible minimum delay timing problems on data signals between different clock regions will be detected.

Similar to the minimum width, the setup and hold times can be set differently for each clock region. This was e.g. done in the application for the communication between the ck_40 region and the ck_160 region, where we used a setup and hold time of 9 ns, which is 1.5 times the clock period of the ck_160 region.

To know if data-clock time violations appear during the simulation, we observe all accesses (read and write) to the data registers. As we simulate per clock region, only 1 clock is active at a particular point in (delta) time. Each register knows by which clock it is driven, and whenever a register is updated to a new value, the time of update is annotated at the register. Because the register is a separate class in our design description, this is easily done inside the class member functions.

The rules that are checked are as follows :

1) A *read* access to a register that belongs to the *active* clock region is always OK.

2) A *read* access of a register with a *different* clock requires a setup-time check, which compares the current time with the update time of the register.

3) A register that has a *write* access, is always driven by the *active* clock. For proper operation the hold-time that was imposed under a different clock, must be checked.

It is clear from this discussion that the minimum width, setup/hold times are not specified or checked at the gate level, but at the clock region level.

V. Results

Besides the simulation and verification possibilities, the C++ design environment also offers HDL code generation. This then completes the design cycle in C++ and establishes the link towards the gate-level implementation. The test-benches and simulation outputs are translated in such a way that they can be used at the gate-level to assure that the behavior is identical before and after logic synthesis.

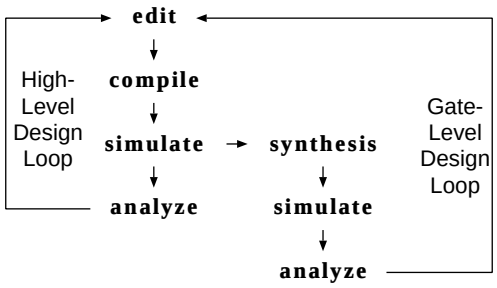

Fig. 6. A High Level Design loop (in C++) is much faster than at the gate level, yielding thus a reduced design time.

Using the analysis possibilities, we are able to have early feedback in the design cycle under the form of a high level design loop : editing source code, compiling it into an intermediate format, simulating the design, checking the results for errors or analyzing the performance. The loop is traversed many times until the results are satisfactory. A second loop is then entered in which the circuits are further synthesized and new results must be analyzed. The main difference between these loops from the designer's point of view is that the 2nd loop takes orders of magnitude more time to traverse than the 1st one.

Our design approach is therefore twofold : make the first loop fast so that iterations are cheap, and reduce the number of iterations in the second loop by analysis or estimation of the results of this loop at higher levels of abstraction, i.e. inside the 1st loop.

VI. Conclusions

The selection of a C++ design environment with a class library for digital circuits, allows to describe an application in a flexible way and to refine it to the architectural RT level. With the newly introduced concepts for the clocks and the clock regions, a combined simulation of event-driven clock signals, data flow blocks and cycle-true FSMDs is made possible.

During the simulation runs, timing checks of clock regions at the RT-level are performed. The main advantage of the presented techniques is thus the ability to raise the abstraction level of gate-level related timing checks to higher levels.

The driving application of the presented techniques was specified using this clock regions approach. The description style, the simulation, the verification and the code generation are generic methods, which makes that they are also available for other applications.

We have used abstraction in C++ to solve a problem which is traditionally attributed to gate level design. This abstraction avoided the use of brute force methods but rather introduced a data model that allows a terse and adequate formulation of solutions to clock region timing problems.

References

[1] K.D.Wagner, "Clock System Design", IEEE Design & Test of Computers, pp. 9-27, Oct. 1988.

[2] Edward A. Lee, "Overview of the Ptolemy Project", ERL Technical Report UCB/ERL No. M98/71 UC Berkeley, 1998.

[3] D. Ziegenbein, K. Richter, R. Ernst, J. Teich, L. Thiele, "Representation of Process Mode Correlation for Scheduling" Proc. of the ICCAD, pp. 54-61, 1998.

[4] E.A.Lee and A.Sangiovanni-Vincentelli, "Comparing models of computation", Proc. of the Int. Conference on Computer-Aided Design, pp. 234-241, 1996.

[5] R.K.Gupta, S.Y.Liao, "Using a Programming Language for Digital System Design", IEEE Design & Test of Computers, pp. 72-80, Jun. 1997.

[6] T.Grötker, R.Schoenen, H.Meyr, "PCC: A Modeling Technique for Mixed Control/Data Flow Systems", Proc. of the DAC, pp. 482-486, 1997.

[7] Joon-Seo Yim, et al., "A C-Based RTL Design Verification Methodology for Complex Microprocessor" Proc. of the DAC, pp. 83-88, 1997.

[8] P.Schaumont, S.Vernalde, L.Rijnders, M.Engels, I.Bolsens, "A Programming Environment for the Design of Complex High Speed ASICs", Proc. of the DAC, pp. 315-320, 1998.

[9] P.Schaumont, S.Vernalde, M.Engels, I.Bolsens, "Low Power Digital Frequency Conversion Architectures", Journal of VLSI Signal Processing, 18, 187-197, 1998.