An Object Oriented Programming Approach for Hardware Design

S. Vernalde, P. Schaumont, I. Bolsens
IMEC, Kapeldreef 75, B-3001 Leuven, Belgium

Abstract

An object oriented programming approach for the design of complex systems in hardware is presented. The paper describes how the usage of object oriented techniques resolves some major obstacles for system-on-chip design. The design of a 10 Mbit/s upstream cable modem is used as a driving example.

1 Introduction

The continuous evolution in ASIC technology allows the integration of complete telecom systems on silicon. This includes the complete information-processing path starting from physical transport of bits, over network layer processing downto user-level multimedia presentation.
Unfortunately not the same can be said from the tools that are needed to design these digital systems. In fact, the tools required to design for instance a mobile phone ASIC, are even far from fitting onto the hard disk of a single high-end Personal Computer.
For one thing, this is due to the fact that each design discipline (like DSP algorithm design, network performance simulation, hardware synthesis and embedded software design) has its own tool and/or favourite environment. As a result, a system level design flow is a patchwork of scripts, translators, and tools.
In addition, much tasks in the construction of an ASIC are still more of an art than a method. Think for instance of bringing a DSP algorithm designed in Matlab to a hardware implementation. There is a multitude of paths leading to a possible solution, and even more of them leading to a dead end.
We observe that there are some major obstacles today to do successful system-on-chip design.

- There is a lack of a single system-level environment that can be used throughout the design flow. While algorithms might be designed at high level in C, gates still have to be synthesised out of HDL. Each manual format translation, no matter how small, is a possible source of errors.
- The designer has insufficient control over the design process. He or she has to accept the result that (synthesis) tools produce. This is a result of those tools being sold as closed boxes. Assembling a system level design flow out of such tools however requires an open environment.
- There is lack of a systematic verification strategy. There are as many testbenches as there are tools used in the design flow. Especially at phases in this flow where drastic changes are done to the design representation (e.g. during the transition from Matlab to Verilog), the development of corresponding or equivalent testbenches is extremely hard – if possible at all.

Being stuck with this situation in several recent demonstrator designs, we turned towards object-oriented C++ technology [6,7]. This allowed us to overcome all of the obstacles that were mentioned. The use of C++ has been demonstrated for the modeling and simulation of parallel hardware systems [1]. Our environment also provides VHDL code generation and HDL testbench generation.
In this paper, an overview of the resulting C++ design environment (called OCAPI) will be given, in addition to the discussion of a concrete design experience. We start by giving a small example of C++ based design, and contrast it to traditional VHDL design. Next, the scope is broadened towards using C++ for a complete system level design flow. This is finally illustrated by the design of an upstream cable modem.

2 Hardware design with C++

When designing a system on chip in C++, we need to take care of both the hardware and software parts of this system. While C++ is a logical choice to devise the software parts, using it for hardware descriptions is less obvious. The match is however closer than one might think at first. The object-oriented capabilities of C++ allow us to write down a representation in terms of objects that closely resemble the actual intended circuit. To illustrate this point, consider the design of a simple

incrementer circuit. It is desired to create a synchronous, digital machine, with one controller sending instructions to a datapath. The descriptions in table 1 show the result in both traditional VHDL coding and in our C++ design environment.

Incrementer in VHDL	Incrementer in C++

```
Architecture RTL of my_processor is
Begin
    SYNC : process (clk)
      Begin

      If (clk'event and
          Clk = '1') then
        Current <= next_state;
        A_at1 <= a;
      End if;
    end process;

    COMB : process (a, a_at1)
       Begin
      A <= a_at1;
       Case current is
        When state1 =>
          a = 0;
          next_state <= state2;
        when state2 =>
          a = a_at1 + 1;
      end case;
      if (reset = '0') then
        a = 0;
      end if;
    end process;
  end RTL;
```

```
#include "ocapi.h"

void main() {
  sig a(ck); // register

  sfg reset; // instruction
  a = 0;

  sfg inc;
  a = a + 1;

  fsm f(ck); // controller
  state state1;
  state state2;

  f << deflt(state1);
  f << state2;

  // state transitions
  state1 << always
         << reset
         << state2;
  state2 << always
         << inc
         << state2;
}
```

Table 1: Comparing equivalent VHDL and C++ descriptions

The circuit has a fairly standard description style in VHDL. We observe however that, by making this VHDL description, the distinction between the controller (finite state machine) and the data processing is lost. In addition, the constructs that are used to write VHDL (processes, variables, case statements, etc) bear little resemblance with the RT-level structure of the incrementer circuit. The C++ description, shown on the right, approaches the description from another side. It uses objects like sfg, state and fsm to reflect the exact design concept that was intented. In this case, sfg creates a datapath instruction, while state and fsm are parts of the controller. All these objects are related to each other through the use of C++ operators and expressions. As these operators execute, an object hierarchy is constructed that reflects the RT behaviour of the processor.
As will be discussed further, we can simulate this object hierarchy and generate VHDL code out of it. But it also allows us to design more effectively.

3 Design flow

In this section, the complete system level design flow is presented. To be able to construct systems on chip in an effective way, the following requirements must be met.

- A SoC design environment must be able to capture behaviour at high level. This is needed for doing algorithmic design and exploration. Therefore, OCAPI initially captures behaviour as a dataflow description, much in the same way as an environment like COSSAP [9] does. One difference is that our system description is a C++ program, where current environments are block-diagram based.
- A SoC design environment must offer the possibility to do detailed design description of hardware, similar to traditional HDL environments. OCAPI includes a set of objects that allow describing hardware at the RT level. In addition, these objects can be co-simulated with the high-level dataflow description.
- A SoC design environment should avoid making manual translations between equivalent design representations. For this purpose, a C++ description made in terms of OCAPI objects can be translated automatically to VHDL. In addition, VHDL testbenches and test vectors are generated

which can be used to repeat simulations in correspondence with the C++ simulation.

- A SoC design environment must support incremental refinement which allows a smooth transition from pure behavioural descriptions downto architecture descriptions. In OCAPI, dataflow and architecture descriptions can be co-simulated. In addition, also floating point and fixed point datatypes can be freely mixed.

The design flow that we use is illustrated in Figure 1. The flow contains three major parts: a system level design part, a hardware synthesis part and a hardware verification part.

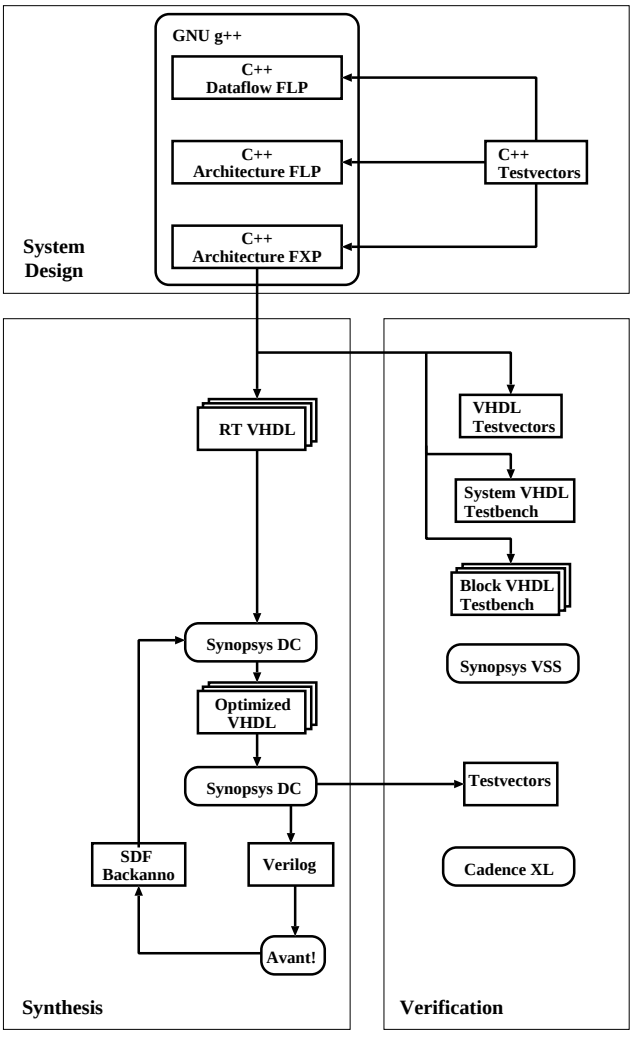

Figure 1: The C++ based system design flow

3.1 System design

The goal of the system design phase is to construct a functional RT-level model of the ASIC under construction. For verification and test purposes, a system level environment model is required. We will use a cable modem receiver example as we go through the entire design flow. For the design of this receiver, the environment model consists of a transmitter model and a channel model. This allows system level simulations and verification of the receiver algorithms. These simulations are collected in a set of C++ testvectors that will be reused in the hardware testbenches.

Initially, a floating-point data flow model of the complete system is constructed (transmitter, channel model, receiver). Next, the receiver is refined to cycle true architecture model. Scheduling the operations of high level descriptions to clock cycles does this. In addition, bringing dataflow to hardware also requires the mapping of the dataflow system-level semantics to an implementation. This is a standard design task for which several solutions exist.

After the architecture has been obtained, the chip signal wordlengths are decided in order to yield a cycle true, bit-true architecture model. Fixed-point refinement is done by means of simulation. The required refinement strategy is dependent on the type of application. However, a good strategy for a digital receiver is the following one. First, a reception quality metric (e.g. constellation purity) is determined using only quantization at the A/D side. Next, the other wordlengths are decided such as to prevent overflow and to maintain the reception quality metric. After these steps, the C++ model is a bit-true clock-cycle true

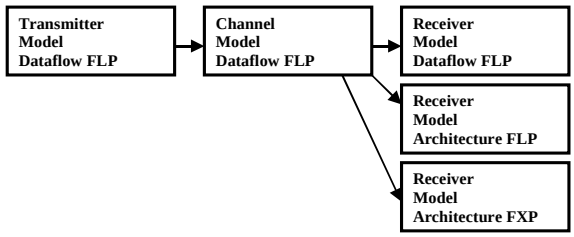

Figure 2: Three possible simulations during system level design of a cable modem receiver

representation of the architecture.

The advantage of using C++ during system design is clear by considering Figure 2. Since OCAPI allows to freely mix algorithmic and architecture descriptions, as well as floating and fixed point datatypes, the receiver description can be co-simulated with the transmitter and channel model at various level of receiver design detail.

The system level design phase concludes with the use of a code generator that creates, out of the C++ description, the input for subsequent hardware synthesis and verification.

- For each block (FSMD) of the receiver, a synthesizable RT-VHDL file is created.
- For the overall chip, a system netlist is generated to connect the various blocks in the design.
- The C++ test vectors are translated into block-level and system-level testbenches. In addition, appropriate testbench drivers are generated.

3.2 Synthesis

The generated VHDL code is directly fed into the Synopsys logic synthesis tools [8]. It is in essence a fully automated process that can be run in batch.
While this synthesis flow is industry standard, the C++ environment and verification that surrounds it is innovative.

3.3 Verification

During synthesis, simulation-based verification is used extensively to track the correctness of the synthesis results. All VHDL-level simulations are done using the generated testbenches with the Synopsys VSS simulator at block-level and system level. The final Verilog netlist is checked using generated production test vectors with the Cadence Verilog-XL simulator.

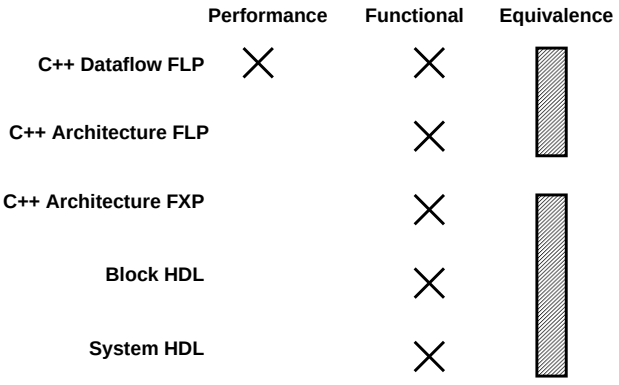

Figure 3: Verification strategies at different levels

Verification is done by C++ simulation during the system design phase and by HDL simulation during the synthesis phase. There are 5 verification levels that correspond to the 5 description levels of the design. Three of them are in C++ (dataflow floating point, cycle-true floating-point and cycle-true fixed point). The remainder two are at VHDL (RT-VHDL and Synopsys-DC VHDL outputs) and Verilog (final netlist) level. The design of testbenches is done in C++, since corresponding HDL testbenches are obtained by code generation. As shown by Figure 3, the test simulations can be categorised in three areas: Performance tests, functional tests, and equivalence tests.
The performance tests are used to check the initial performance of the design. For a digital receiver, test scenarios include varying levels of channel noise, phase distortion, carrier frequency deviation, amplitude slope distortion, gain variation and burst spacing. These tests ensure that the initial algorithmic model has the desired performance.
The functional tests check the correct operation of a design within one verification level. Typical tests include for instance the reception of a known data sequence. The goal of these tests is to perform a simulation with maximal coverage of the design description.
Equivalence tests compare the operation of one level to the next. They are applied at either floating-point level or else fixed-point level. Equivalence tests do a one-to-one comparison of values on the system interconnect at corresponding time-points.

4 An upstream cable modem

Using the OCAPI object library and the design flow discussed above, we have completed several demonstrator designs including an upstream cable modem, a DECT transceiver and a MPEG-4 image coder. Some details on the cable modem design are included here to illustrate the power of C++ based design.
This upstream cable modem was developed in a research project in co-operation with Siemens-Atea, Belgium [5]. An upstream cable modem as shown in Figure 4 resides at the head-end of the HFC television access network.

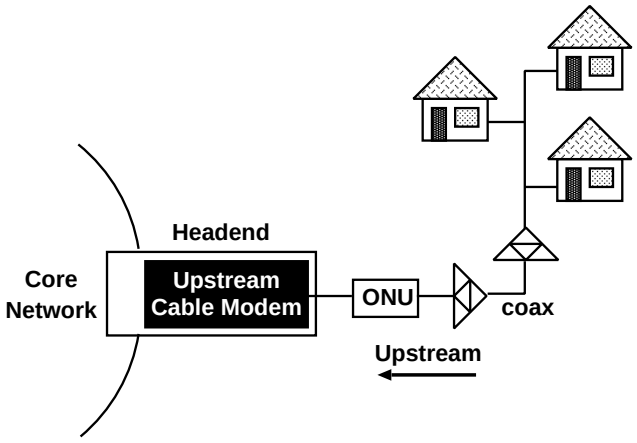

Figure 4: HFC network architecture with upstream cable modem

HFC is a network architecture that is built up of coax and fiber. The low-loss fiber part connects a head-end to an optical network unit (ONU). In the coax part, regularly spaced distribution and upstream amplifiers maintain the signal level. In upstream HFC communications, modulated data is transmitted from the consumer side to the head-end. The single chip digital receiver that we have developed is embedded in the head-end and demodulates this data. It takes care of the physical layer signal processing required for QAM16 or QPSK modulation. Offering 10 Mbits/s data throughput in a 3.3 MHz upstream channel band, it is compliant with the main HFC communications standards (MCNS/DOCSIS [2], DAVIC/DVB [3] and IEEE 802.14 [4]). The chip relies extensively on digital signal processing to demodulate and decode upstream signal bursts. It also estimates and automatically corrects various transmission impairments occurring on the upstream HFC channel. Such impairments include varying signal levels, carrier frequency deviations, group delay distortion and amplitude variation. In addition, the chip provides an interface for an external Reed Solomon channel decoder that combats channel noise effects.

4.1 Development

The chip was developed from scratch with the C++ based design flow. Therefore we started by constructing a system level functional model in C++. This model includes a burst transmitter, a channel model, and a receiver functional dataflow model. Such a system model allows to explore various receiver algorithms and to construct system level testbenches that determine the overall system performance.

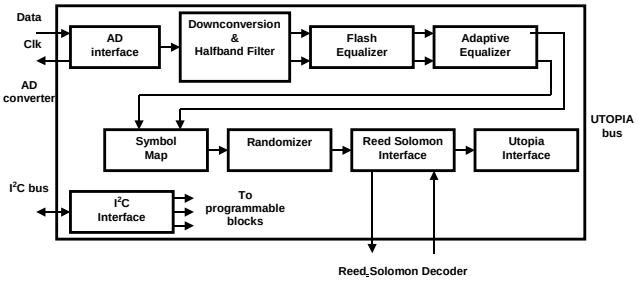

Figure 5: Upstream cable modem chip architecture

The resulting receiver, shown in Figure 5, has been optimised for minimal communication overhead. For this purpose, it contains an advanced flash equalisation algorithm that works with a short, fixed length preamble independent of the communication channel conditions. It also makes the core particularly suited for multimedia applications, in which wide ranges of bitrates need to be supported. All signal processing is done entirely digital, making the performance reliable, predictable and free of tuning. This signal processing is performed in a chain of independent blocks. This complex allows receiving QAM16/QPSK burst signals with an interburst spacing of only 4 symbols and a preamble of 17 symbols.

Care was taken to make the functionality programmable. An I2C programming interface allows in-the-field configuration and adjusting of demodulation parameters. The burst payload length is programmable. In addition, various parameters such as received signal power and equaliser coefficients can be extracted for signal quality estimation.

The complete development process starting at algorithm design and ending with a clock-cycle true, bittrue architecture was done in C++ with the OCAPI library. Once the architecture model was available, synthesisable RT-VHDL code was generated to bring the receiver circuit to a gate level implementation. The code size statistics of the chip in the subsequent design phases are shown in table 2. They illustrate the compactness that can be achieved by using object orientation.

Specification		
C++ Dataflow	922	lines
C++ Architecture	4426	lines
RT VHDL	21798	lines
Gate Level VHDL	154952	lines

Table 2: Cable modem code size

Using C++ based design we experienced tight control over the entire design flow. This is because there is only a single environment in which system level design was performed, including the development of testbenches. We ensured that the C++ simulations were correct, after which only the equivalence between C++ and generated HDL had to be shown. This was straightforward since the testbenches are generated from the C++ design too. The design flow has also yielded a dual result. At one hand, we have the silicon, which is being integrated in a head-end system. At the other hand, we also have the C++ software that yielded the receiver. This code is now available as a software package that can perform behavioural high level simulation, and generate synthesisable VHDL code. The C++ code is highly parameterised and allows tuning wordlength specifications as well as architectural parameters. The package also includes an extensive test script that is used to verify correct operation for a chosen set of parameters. Such a package is an ideal embodiment of an IP-core. A flow for such a core is shown in Figure 6.

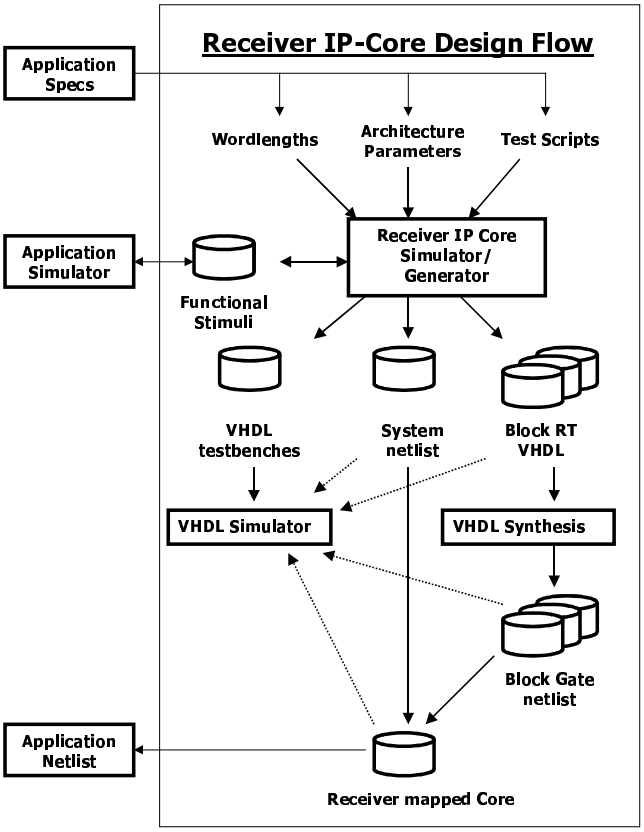

Figure 6: IP-core design flow

5 Conclusions

A design methodology for systems on silicon was presented that is based on object oriented programming techniques. It was shown that the application of object oriented programming to hardware design allows to alleviate some of the major obstacles for efficient system level design:

- There is a single system-level design environment for both algorithm design and architecture design. The transitions between the two levels are done by incremental refinement of descriptions.
- By using a programming language, the designer has full control over the design process. This open environment provides a very effective way to deal with the diversity and heterogeneity of system-on-chip design.
- Because of the open design environment, verification is supported throughout the design process. Three types of verification were identified.

The design of an upstream cable modem using this methodology has resulted in a short design time and first time right silicon.

Acknowledgements

This work was carried out under the Flemish Impulse Program for Information Technology. For the realisation of the cable modem, we would explicitly like to thank Siemens Atea for their effort in the collaboration.

References

[1] R. K. Gupta and S. Y. Liao. Using a programming language for digital system design. IEEE design and Test of Computers, pages 72 – 80, April-June 1997.

[2] Data Over Cable Service Interface Specifications (DOCSIS) MCNS. http://www.cablemodem.com.

[3] DAVIC 1.3 part 8 Digital Audio Visual Council. http://www.davic.org.

[4] IEEE 802.14 Cable TV Protocol Working Group. http://walkingdog.com/catv.

[5] Siemens Atea R&D Technology Homepage. http://www.siemens.be/atea/products_services/rd_technology/rd_frames.htm.

[6] P. Schaumont, S. Vernalde, L. Rijnders, M. Engels and I. Bolsens. A programming environment for the design of complex high speed asics. In Proc. DAC 1998.

[7] P. Schaumont, S. Vernalde, M. Engels and I. Bolsens. Synthesis of multi-rate and variable rate digital circuits for high throughput applications. In Proc. EDTC 1997.

[8] Synopsys Inc., 700 E. Middlefield Rd, Mountain View, CA 94043. DC User's Manual.

[9] Synopsys Inc., 700 E. Middlefield Rd, Mountain View, CA 94043. COSSAP User's Manual.