Design and Verification of a Processor Using VHDL, Verilog, SystemC, and C++

Dr. Greg Tumbush, Starkey Labs, Colorado Springs, CO
Bill Dittenhofer, Starkey Labs, Colorado Springs, CO

Introduction

The processor designed at Starkey Labs was a general-purpose, n-bit, pipelined, programmable digital signal processor (DSP) targeted for low-power applications, such as hearing aids. The individual strengths of SystemC, C++, Verilog, and VHDL were utilized in the design and verification of this DSP. A SystemC model of the DSP was developed to examine architectural trade-offs and serve as a verification vehicle. C++ was used to develop several support tools, including assemblers and disassemblers. Verilog was used for the final, synthesizable model. And a single VHDL testbench was used to verify the SystemC model and compare it to the Verilog model. ModelSim® was the simulator used for this project. Initially a home-grown Foreign Language Interface (FLI) was developed and used at Starkey Labs until native SystemC support with built-in SystemC Verification (SCV) library support became available in ModelSim 5.8. The FLI enabled two-way communication between the VHDL testbench, Verilog RTL model, and the SystemC model. Due to the efficiency of this environment, two engineers were able to design and verify the DSP in a short time.

This paper examines useful techniques for verifying an RTL design against a SystemC model. The I/O and points within either model were passed through the FLI on a cycle-by-cycle basis. Points of interest were compared between the SystemC and RTL to produce an inherently self-checking environment. Pulling points through the FLI also enabled a block to be developed in-stasis before the interface logic was designed. We call this "putting the block on life support". This allowed block development to proceed rapidly. As the design matured, the block ran without assistance from the SystemC model.

Verification of the DSP was approached from four directions, as illustrated in Figure 1: block-level directed, processor directed, random tests, and ASIC emulation with an FPGA. Each direction provided verification capabilities that complemented the strengths and weaknesses of the other in areas such as interfaces, instructions, algorithms, and listening tests. Because a robust assembler/disassembler was also developed along with the DSP, test cases were written at the instruction level and debugged efficiently. At this level of abstraction, test cases were quickly developed and checked for correctness. Also, the same tests run at the block level were run at both the processor level and in the FPGA, facilitating test reuse. Random tests were generated using the SystemC Verification (SCV) library at the instruction level. An infinite number of instructions were randomly generated and run on the DSP, emphasizing particular classes of instructions. Lastly, algorithms developed on the SystemC model were ported to the FPGA, and actual listening tests were performed.

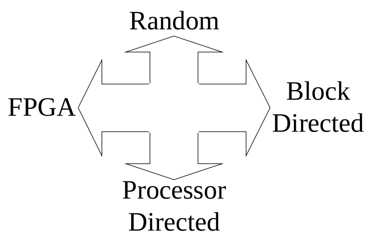

Figure 1. Four Level Verification Approach

A Multi-Language flow

A multi-language design and verification environment allows the strengths of many languages to be leveraged. In our environment, the linker, assembler, and disassembler for DSP code was written in C++. C++ was leveraged due to its wide user base as well as its appropriateness for this application. C++ can be developed and simulated with low-cost software on a low-cost system in an environment familiar to software developers.

SystemC adds the notion of time, concurrency, and hardware data types – all of which are lacking in C++. SystemC was utilized to build a high level but still cycle and pin accurate model of the DSP. This allowed architectural tradeoffs to be examined and provided a vehicle for co-simulation of the upcoming RTL. Additionally, algorithm developers did not have to wait for the RTL code to be completed.

It is important to determine the appropriate level for the SystemC model. Is cycle accurate important? Is pin accurate important? For the model of an algorithm, such as an FFT, FIR, etc., a non-cycle accurate but pin accurate model can be developed. A FIFO is inserted after the SystemC model to throttle the comparison of results. The RTL model determines when a result is available and draws a result from the SystemC results FIFO. The results are then compared. This is known as Kahn modeling and is illustrated in Figure 2. A processor model, on the other hand, must be cycle and bit accurate. It may or may not be pin accurate.

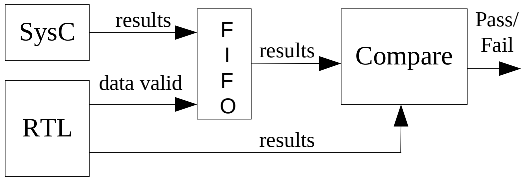

Figure 2. Kahn Modeling

VHDL was used for the following purposes: testbench development, a communication interface through the FLI with SystemC, and comparison of SystemC signals with RTL. ModelSim *signal_spy* constructs were used for introspection into the Verilog RTL signals[1]. A decision needs to be made whether to compare the signals in SystemC or VHDL. In SystemC, only the *sc_logic* type is multi-valued and can represent an "X". Simulations will be extremely slow if all SystemC signals are of this type. A signal that is an "X" in RTL can positively compare to a SystemC signal not of *sc_logic* type if the comparison is done in SystemC[2]. Therefore, comparisons should be done in VHDL.

The records and overloading features of VHDL were used extensively. All signals compared were encapsulated into a record that contained the RTL value, SystemC value, error count, and error color. The error count is a running total of the number of errors for this signal. The error color provides a quick visual check of the comparison status. Red indicates a mismatch in the present cycle while green indicates a match. This proved indispensable for quick debugging.

Finally, Verilog was utilized to develop the synthesizable RTL. The available engineering base at Starkey used Verilog on previous projects. Additionally, there was a large pool of Verilog literate engineers available in Colorado, facilitating recruitment. All of the engineers working on the project were recent hires.

It should be emphasized that all of the work involved in developing the SystemC model, assembler, linkers, etc. would have been done whether or not an ASIC was ever produced by Starkey Labs. This work was necessary to enable algorithm developers to examine performance tradeoffs and debug the final processor. Considering that this project was completed by a two-man team in a short period of time, leveraging existing components was of the utmost importance.

Verification Environment

A high-level diagram of the verification environment is represented in Figure 3. The reference system is the model of the DSP written in SystemC. From this model the I/O and any internal signals can be brought through the FLI and observed. The design under test (DUT) is the synthesizable RTL model written in Verilog. Similarly the I/O and internal signals of this model may also be observed. The important difference is that these signals will not be brought through the FLI. With few exceptions, the FLI is used to observe SystemC signals. The reference results and DUT results are compared in the results comparator. The results comparator observes the signals and flags an error if they do not compare. Various monitors and checkers may be bolted onto the RTL to provide further checking, such as adherence to a bus protocol. Lastly, stimulus is provided to both the SystemC model and DUT simultaneously and in parallel.

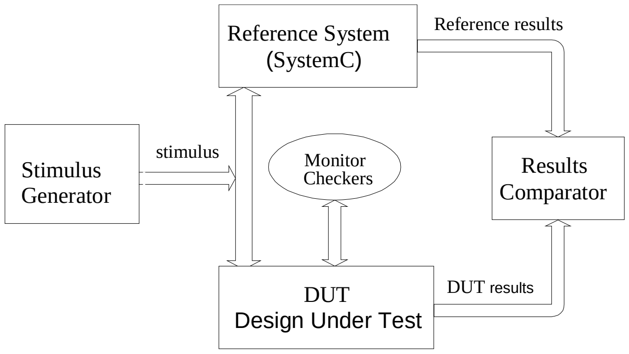

Figure 3. Verification Environment

The FLI

The design process at Starkey consisted of comparing a C-based algorithm model to RTL. This was accomplished by generating vectors from each independently running kernel (RTL and C), storing the results of each and then comparing the databases against one another. An illustration of this flow is in Figure 4. This led to very high overhead for storage and comparisons. This worked for 100,000 gate projects but the next generation would be much larger. A new, more efficient method was needed.

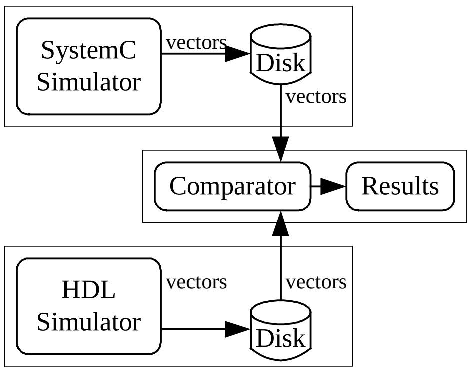

Figure 4. Running SystemC and HDL Kernel Independently

Running both simulators concurrently in the same kernel and doing the compares cycle by cycle in memory were found to be much more efficient. Our tests showed conservative improvements of 50X. This allows for lower storage (VCD files) and lower CPU requirements (post processing comparison). As an added bonus, the system is much more flexible for debug. An illustration of this flow is in Figure 5. Note: Everything runs in memory.

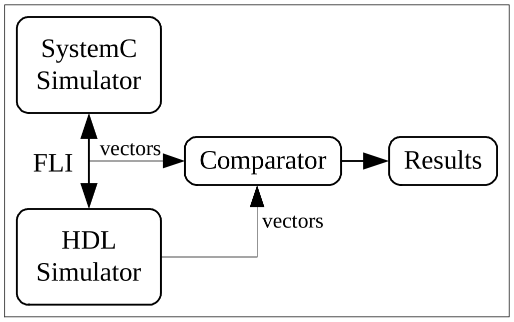

Figure 5. Running SystemC and HDL Kernels Concurrently

At the time, direct compile of the SystemC kernel into ModelSim was not available. In order to manage risk, Starkey developed an FLI based interface for importing, controlling, and communicating with the SystemC kernel. Essentially, it allows us to run the SystemC based models and simulator in ModelSim along with Verilog and VHDL. This enables comparison of internal SystemC model variables with counterparts in Verilog on a cycle-for-cycle, bit-for-bit basis. This also takes advantage of the SystemC internal bit types, versus “masking” bits in a vector in C based methods.

Being able to expose internal variables is a tremendous boon to debug. As we found difficult to debug areas in the design, the SystemC modelers would expose the internal variables in the model for comparison with the RTL. The testbench could then detect, for example, bad addresses when data is written to memory. This often flagged a problem long before a corresponding read to the same address exposed the errant write. Comparisons were perfomed on approximately 65 signals split up between registers, memory interfaces, I/O, and critical signals.

We were able to reuse C++ code from our system model for a disassembler. This allowed us, via the FLI, to display instructions, with operands, in an easily readable text format versus hex, as illustrated in Table 1. The ability to determine what the processor is doing at a glance is very important.

Table 1. Converting Hex Instructions to Text

Cycle Count	255	256	257
Instruction in hex	4ca5a	10fca	72f22
Instruction in text	XFR AX1, C0	MUL ACA, XA	RI_PO

Additionally, this method allowed us to build a divide and conquer approach to the design as depicted in Figure 6. Normally, one designs an internal block of RTL, builds a testbench, and debugs the block in isolation. When ready, it is connected in the system, and the testbench and vectors lose their relevance. Our approach was to build the minimum necessary components to support a block: those being the program controller and the data path sections. These were supplied with input from the SystemC pipeline, which was known to be correct for interrupts, jumps, etc. The rest of the stimuli for the internal block came from the internal variables via the FLI (direct compile would be used today). The output from that RTL block is compared to the SystemC model, and one can observe any differences. This sped up implementation and verification of each core, as individual testbenches were not needed. An additional benefit is that the tests were written in assembler, which was reusable in the next level of unit testing.

Today, direct compile is available for ModelSim. Our tests have shown it to be a very flexible, convenient method of importing SystemC models into ModelSim. Using direct compile, the SystemC model does not have to be written with accessor functions to points of interest within the model. The SystemC model is viewed as just another HDL and can be probed as such.

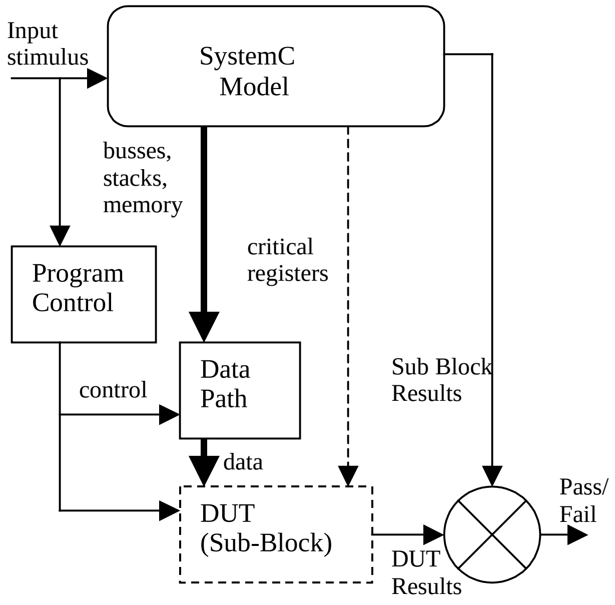

Figure 6. Divide and Conquer Approach to Design

Verification of the DSP

A DSP’s primary focus is to execute instructions. Therefore, the main focus of verification is to stimulate the SystemC and RTL with a wide range of instructions. An instruction is made up of a mnemonic and 0,1, or 2 operands. A mnemonic is the operation to be performed, such as transfer, add, and multiply. The operand is the register or memory location that the mnemonic will be performed on.

Due to the availability of an assembler and linker (written in SystemC), most testing can be performed by simply developing sequences of instruction. Testing at such a high level lends itself easily to random generation of instructions. Each mnemonic is enumerated, and the allowable operands for the mnemonic are constrained. Then the *next* function of the SCV Library is used to generate a new instruction[3]. The number of instructions that can be simulated in a single test is only limited by the size of program memory. An advantage of SystemC and C++ is that functions can execute in zero RTL simulation time. This fact can be leveraged to randomly generate a new set of instructions, assemble and link the instructions, and load program memory in zero simulation time. In this manner an infinite number of instructions can be generated and executed.

The State of the DSP

A challenge of verifying a DSP is the definition of correctness. That is, what SystemC signals must match what RTL signals at what time? Does every single internal signal have to match on every cycle, or is there a subset of both signals and time that can be defined? For the DSP design at Starkey Labs, it was determined that all internal registers as well as the I/O would be compared. Internal registers are those defined in the specification, not those produced by synthesis. This criteria would define the *State* of the DSP. Some I/O, such as memory, would only be compared when the enables are active. Data and address lines are allowed to miscompare when not active. If all comparisons are positive the DSP is deemed to be operating correctly.

Block-Level Verification

As soon as a few core blocks were completed they could be verified by putting them on “life-support” from the SystemC model. For example, a program decoder could be fed instructions from the SystemC model before the program controller was developed. A diagram of this concept is in Figure 7. In this manner individual blocks could be verified, simplifying integration debug immensely. After the peripheral blocks (in this case the program controller) were finished the “life support” was removed, and the RTL ran independently. Only rudimentary testing is done at this level, and all tests are directed (i.e. not random).

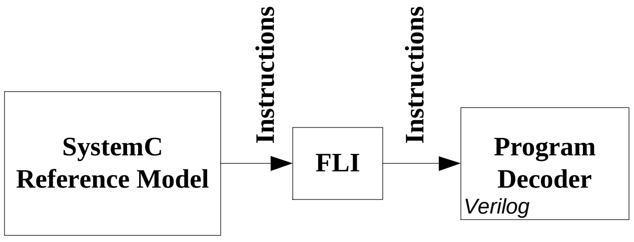

Figure 7. Life Support from SystemC Model

Processor-Level Verification

At the processor level, all the tests developed at the block level can be directly reused. The only difference is the RTL now runs standalone. Verification at the processor level is primarily through random testing. Directed tests are written at the processor level to verify the interfaces of the DSP. Additional directed tests are also written to achieve the desired code coverage. A detailed block diagram of the final verification environment is depicted in Figure 8.

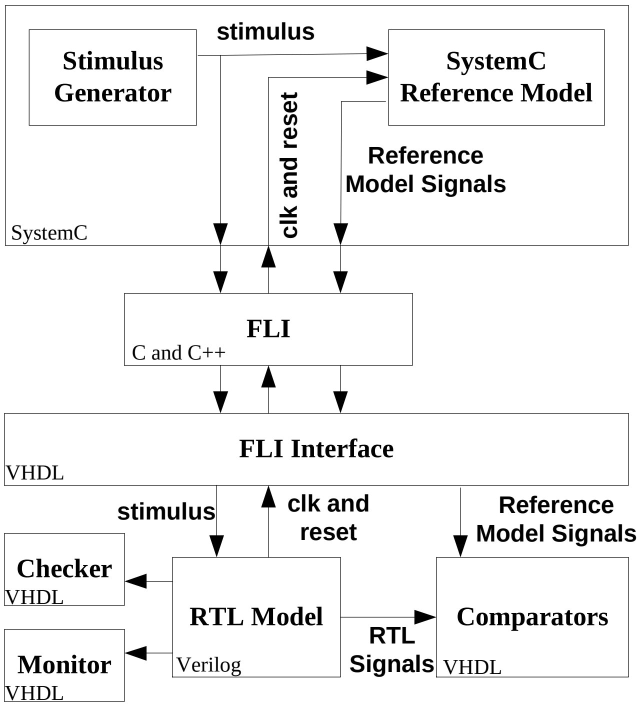

Figure 8. Processor Level Verification Environment

Random Instruction Generation

Using the SCV Library, massive amounts of instructions can be generated in a very short period of time. However, unless the instructions are constrained correctly, the time-savings will be lost to the time spent debugging illegal instruction sequences. The efficiencies of random verification techniques enabled a single engineer to verify the DSP as well as participate in the design effort.

Achieving acceptable code coverage results with directed tests only would have been impossible with our small team. Table 2 illustrates code coverage results (as a percentage) with directed tests only, random tests only, and combined tests. Comparable code coverage results were obtained with random verification in a much shorter period of time. However, directed testing is still vital to hit important facets of the design. Directed tests are also much easier to debug.

Table 2. Code Coverage Results (%)

Test's	Statement	Branch	Condition
Directed	84.0	71.9	65.7
Random	81.0	65.2	62.7
Combined	90.8	79.5	73.5

An example of random instructions generation follows. Suppose we have a processor that has only three instructions, COM (compare), ADD (add), and SUB (subtract). Operand 1 is allowed to be one of four registers, X0, X1, X2, and X3. Operand 2 is allowed to be one of two registers, Y0 and Y1. Below are Random_Instruction.cpp and Random_Instruction.h.

Random_Instruction.cpp line 4 is the entry point from the SystemC library to the users's code. Line 8 creates an output file of instructions. Lines 9-11 create arrays of strings for the pair of Operands and the OpCode. We will randomly generate a constrained index into these arrays to create the instructions. Line 12 creates an object of type *enMNEMONICS* to randomize upon. Line 13 constructs the constraint and gives it a name of *add.* Line 16 generates a new OpCode. Line 19 generates a new pair of operands. Finally, line 20 calls the function to generate the instruction

```
1  #include <Random_Instruction.h>
2  #include <iostream>
3
4  int sc_main(int argc, char** argv)
5  {
6    int i;
7
8    ofstream OutFile("instructions.test", ios::out);11
9     char* Operand1Str[] = {"X0", "X1", "X2", "X3"};
10    char* Operand2Str[] = {"Y0", "Y1"};
11    char *OpCodeStr[] = {"ADD", "SUB", "COM"};
12    scv_smart_ptr<enMNEMONICS > OpCode;
13    Add_Constraint add("add");
14    for (i=0;i<8;i++)
15        {
16        OpCode->next();
17        if ((*OpCode == ADD) | (*OpCode == COM) |
(*OpCode == SUB))
18           {
19             add.next();
20           generate_ADD(OutFile, OpCodeStr[*OpCode],
Operand1Str[*add.Operand1],
Operand2Str[*add.Operand2]);
21         }
22    }
23  return 0;
24  }
25
26  void generate_ADD(ofstream &OutFile, char *OpCode,
char *Operand1, char *Operand2)
27  {
28   OutFile << "    " << OpCode << " " << Operand1 << ",
" << Operand2 << endl;
29  }
```

RandomInstruction.h lines 1 and 2 includes the SystemC and SCV header files. Line 4 is the functional prototype of the function that outputs the instruction. Line 5 creates an enumerated type called *enMNEMONICS.* Lines 7-16 create a partial template specialization of *scv_extensions* for enumerated types. Line 18 declares a structure *Add_Constraint* containing the elements *Operand1* and *Operand2* of type *unsigned int* to randomize upon. Lines 21-24 uses a constructor to constrain the indexes *Operand1* and *Operand2.*

```
1   #include <systemc.h>
2  #include <scv.h>
3
4  void generate_ADD(ofstream &, char *, char *, char *);
5  enum enMNEMONICS {ADD, SUB, COM};
6
7  template<>
8  class scv_extensions<enMNEMONICS> : public
scv_enum_base<enMNEMONICS> {
9  public:
10
11   SCV_ENUM_CTOR(enMNEMONICS) {
12     SCV_ENUM(ADD);
13     SCV_ENUM(SUB);
14     SCV_ENUM(COM);
15      }
16  };
17
18  struct Add_Constraint : public scv_constraint_base {
19   scv_smart_ptr<unsigned int > Operand1;
20   scv_smart_ptr<unsigned int > Operand2;
21   SCV_CONSTRAINT_CTOR(Add_Constraint) {
22     SCV_CONSTRAINT(Operand1() < 4);
23     SCV_CONSTRAINT(Operand2() < 2);
24       }
25     };
```

The output of this code is:

```
SUB X0, Y0
ADD X3, Y1
ADD X2, Y0
SUB X0, Y1
ADD X0, Y1
ADD X1, Y1
COM X1, Y0
SUB X3, Y0
```

FPGA Verification

Further verification will be done with an FPGA-based ASIC emulation platform. The ASIC emulator excels in testing interfaces and other characteristics of the DSP not easily exposed through instruction-level testing. In addition, an algorithm can be loaded onto the DSP and real listening tests performed. This will require various synthesizable peripherals to be available. A challenge for ASIC emulation is the recreation of a failure within a virtual environment where it can be easily debugged. This may require extensions to the testing environment or deferral of the test to the system utilizing the DSP. It is vitally important to have a high level of confidence in the RTL to avoid extensive debugging time.

Concluding Remarks

We leveraged four different languages to design and verify the DSP.

1) C++
    a) Developed linker, assembler, and disassembler.
    b) Inexpensive to develop, maintain, and test.
    c) Quick development time.

2) SystemC
    a) Allowed us to leverage DSP algorithms done in C++.
    b) Allowed us to make reference models bit and cycle accurate.
    c) Allowed use of directed random verification (DRV).

3) VHDL
    a) Allowed us to build watchers and monitors in an HDL environment.
    b) Allowed use of FLI for better performance (versus PLI).
    c) Abstract types, records, and overloading made for useful displays in the simulator.

4) Verilog
    a) CaseX and CaseZ were very useful for instruction decoders.
    b) Allowed us to leverage existing RTL designers in Colorado.

5) Additionally, the ModelSim simulator gets kudos for:
    a) SignalSpy eased probing of internal RTL for monitoring and comparison.
    b) Dual language feature was seamless and easy to use.
    c) FLI turned out to be very useful and powerful.
    d) Direct compile adds ease of use to SystemC models.

We used one environment which allowed us to rapidly verify the DSP from a block to a processor level. This allowed us to leverage the same tests (assembler code) at both the block and processor level.

The use of a cycle and bit accurate reference model in SystemC eased debug and verification. It also allowed us to make extensive use of DRV, a real labor saver. An additional benefit was the improved validation of the reference model as it went thru the process. This model will be used in several system level simulators, some with RTL and some without. All will benefit from the accuracy of this model.

The overall architecture made for a cost effective solution. The major cost was for ModelSim SE simulators, a cost we would have borne anyway. Running SystemC inside ModelSim (with the FLI as the communication channel) and doing the comparison on the fly in memory is the most efficient method. We were able to reduce our expected server / simulator license ratio by 50 percent. Regressions are faster as no disk I/O is needed unless a failure is detected and a waveform file created. For a small company with a limited tools budget, these efficiencies were most welcome.

References

1. Model Technology (Mar 2003), “ModelSim SE Command Reference, Version 5.7c”.
2. Open SystemC Initiative (2003), “SystemC 2.0.1 Language Reference Manual, Revision 1.0”.
3. Members of the SystemC Verification Working Group (Dec 2002), “SystemC Verification Standard Specification, Version 1.0b”.