This article describes an automated approach to improve design coverage by utilizing genetic algorithms added to standard UVM verification environments running in Questa® from Mentor Graphics®. To demonstrate the effectiveness of the approach, the article will utilize real-world data from the verification of a 32-bit ASIP processor from Codasip®.

**INTRODUCTION**

Application-specific instruction-set processors (ASIP) have become an integral part of embedded systems, as they can be optimized for high performance, small area, and low power consumption by changing the processor’s instruction-set to complement the class of target applications. This ability makes them an ideal fit for many applications — including the Internet of Things (IoT) and medical devices — that require advanced data processing with extremely low power.

The ASIP used in this example was modeled using Codasip® Studio, a highly integrated development environment that allows rapid ASIP prototyping and development. Using Codasip Studio, the instruction set of an ASIP — the instruction-accurate model (IA) — and the behavior of that ASIP at the hardware level — the cycle-accurate model (CA) — are described in a processor description language called CodAL™. Using this description, everything needed for implementation and integration of the ASIP is automatically generated; including the HDL representation of the ASIP (in Verilog, VHDL, or SystemC), a complete ASIP toolchain based on LLVM/GNU (compiler, simulator, debugger, profiler, assembler, and libraries), a virtual prototype (SystemC and QEMU), and the UVM based verification environment.

The power and flexibility of ASIPs, however, raises some important verification issues. One concern is the verification of the generated HDL description of the processor with respect to the reference model (e.g., the pipeline behavior, communication with a memory, etc.). Another is the verification of the ability of the ASIP to execute software applications (sequences of instructions) correctly. In practice, verification is achieved by evaluating a large amount of applications (programs) in simulation. These applications originate from various benchmarks/test suites and can be manually written or automatically generated.

**UVM GENERATION WITH CODASIP STUDIO**

Codasip Studio automates the generation of UVM environments for execution in the Questa® simulator from Mentor Graphics®. In this environment, the HDL representations of either individual ASIPs or complex platforms consisting of several ASIPs, buses, memories, and other IP components act as the design under verification (DUV). The reference model, which is a very important part in the UVM architecture, is automatically generated from the high-level IA model of an ASIP and from C++ models of external IP components. A generator of random applications for the target ASIP is included in the generated ASIP toolchain and will be used to generate stimuli for ASIP verification. For illustration, see Figure 1.

An important feature of the generated UVM environment is the optimization algorithm that runs in the background and adjusts the constraints of the random application generator in order to automate and speed up coverage closure. This optimization reduces the effort needed to prepare comprehensive verification stimuli and improves verification productivity. The remainder of this article will look at how this UVM environment was structured and how it has been used for real-world ASIP verification.

**THE EXPERIMENTAL ASIP**

Codix-RISC™ is a 32-bit processor with six pipeline stages. It contains 32 x 32b general purpose registers with up to three read and one write ports by default. Hazards are handled by the generated LLVM compiler, although hardware hazard management is an option. Codix-RISC is suitable for embedded signal, video, and wireless processing applications that include...
some application specific code, such as video codecs, malware detection, security algorithms, and any other computationally demanding low-power tasks. In the standard verification of Codix-RISC, different test applications are evaluated. Initially, benchmark applications and language test suites are used; then applications from the random application generator are applied.

PROPOSED UVM OPTIMIZATION TO IMPROVE COVERAGE

Coverage metrics tracked for Codix-RISC in the standard UVM-based environment are: code coverage (statement, branch, expression, FSM), functional coverage (requests, status, and responses on the bus interface), and instruction coverage (the complete instruction set and sequences of instructions).

In the previously described standard verification, no feedback is provided to the generator about the achieved coverage. In the random approach, the generator just produces random applications; therefore, to achieve reasonable coverage, a considerable number of applications must be applied to the ASIP. To optimize this approach, we incorporated a genetic algorithm (GA) as the main optimization tool, which provides feedback about achieved coverage and modifies the constraints of the generator. In particular, the additional constraints restrict the size of applications (100–1000 instructions) and define the probability that each instruction from the instruction-set will be generated. The generator works with its original basic set of constraints, plus the probability and size constraints, the values of which are modified by the GA.

GA employs a population of candidate solutions that evolve through several generations. The quality of candidate solutions is determined by a fitness function. According to the fitness function, the best solutions are selected and serve as parents for the next generation. New candidate solutions are created by genetic operator mutation and crossover. If the algorithm and its parameters are tuned well, the average fitness of the population improves over time. This means the algorithm is spending effort in exploring profitable parts of the search space. At the same time, genetic operators ensure diversity, so the algorithm is resilient to the problem of local optimum. For coverage-driven verification, GA serves unconventionally as an optimizer that runs in the background of the verification process. This means that in contrast to the typical application of GA, it is not intended only for searching for good candidate solutions. Moreover, as profitable values of GA parameters are found, additional tuning is no longer required. These parameters have been tested during verification of several ASIPs, as well as other IP components, and resulted in a stable level of optimization.

The GA-driven approach is provided as an extension of the basic functional verification environment prepared according to the UVM and follows the principle of object oriented programming (OOP). Our aim was to integrate the GA components effectively so that interference to the standard architecture of UVM is minimal. Figure 2 highlights the SystemVerilog components/classes that are added to the standard UVM environment.

Figure 2. The UVM environment with GA components.

The GA component represents the core of the algorithm. It produces chromosomes (coding representations of candidate solutions) with encoded constraints that are passed through the Chromosome Sequencer and then propagated to the random application generator. The generator then generates one application per chromosome. For the Codix-RISC processor, generated applications are directly loaded to the program memory of the processor.
using the Application Loader component, and the verification in simulation is started immediately.

The following is an example of simplified UVM source code of the `CodixRiscChromosome` class, and of the `run_phase` of the `CodixRiscTest` class.

class CodixRiscChromosome;
    real fitness;   // Fitness value of chromosome
    rand bit[6:0] chromosome_parts[];  // Constraints for the generator

    function void crossover( inout CodixRiscChromosome chrom );
        // Temporary chromosome
        CodixRiscChromosome tmpChrom = new();

        // Position of crossover
        int main_pos = $urandom_range (CONSTRAINTS_NUMBER);
        tmpChrom = chrom.clone();
        chrom.chromosome_parts[main_pos] = this.chromosome_parts[main_pos];
        this.chromosome_parts[main_pos] = tmpChrom.chromosome_parts[main_pos];
    endfunction: crossover

    function CodixRiscChromosome mutate( int unsigned maxMutations );
        // Position of mutation
        int main_pos;

        // Number of mutations
        int mutationCount = $urandom_range (maxMutations);
        for (int i=0; i < mutationCount; i++) begin
            // Position of crossover
            main_pos = $urandom_range (CONSTRAINTS_NUMBER);
            // Make bit flips at the position
            bitflips(this.chromosome_parts[main_pos]);
        end
    endfunction : mutate

class CodixRiscTest extends CodixRiscTestBase;
    // registration of component tools
    `uvm_component_utils( codix_risc_platform_ca_t_test )

    task run_phase( uvm_phase phase );
        CodixRiscChromosome best_chromosome;
        Population p, new_p;
        int counter = 0;

        p = createOrLoadInitialPopulation();

        // Evaluate population in simulation
        evaluatePopulation(p);

        // Get best chromosome
        best_chromosome = getBestChromosome(p);

        while (counter < NUMBER_OF_GENERATIONS) begin
            // create new population
            new_p = new();

            // check elitism
            if (ELITISM) new_p[0] = best_chromosome;

            // select parents and create new chromosomes
            new_p = selectAndReplace(p);

            // Evaluate new chromosomes in simulation
            evaluatePopulation(new_p);

            // Get best chromosome
            best_chromosome = getBestChromosome(new_p);

            // create final population
            p = createNewPopulation(p, new_p);

            counter ++;
        end
    endtask: run_phase

class CodixRiscTest;
We compared the effectiveness of the GA optimization to two standard approaches. In the first approach, the benchmark applications are used for verification of the ASIP. In the second approach, the random application generator is used. The proposed GA-driven approach also uses the random application generator but adds additional constraints to the generator encoded in the chromosome. The graph in Figure 3 demonstrates the results of these three approaches. It compares average values from 20 different measures for each approach (using various seeds). The x-axis represents the number of evaluated applications on the Codix-RISC processor, and the y-axis shows the achieved level of total coverage.

The average level of coverage achieved by 1000 benchmark applications was 88 percent. The average level of coverage achieved by 1000 random applications was 97.3 percent. In the GA-driven approach, the random generator produced random applications for 20, 50, and 100 generations (the 100 generations scenario is captured in the graph). The size of the population was always the same: ten chromosomes. For the test with 20 generations, the average level of coverage was 97.78 percent. For the test with 50 generations, the average level of coverage was 98.72 percent. For the test with 100 generations, the average level of coverage was 98.89 percent. The results show that for all tests utilizing GA optimization, we were able to achieve better coverage than with standard approaches. It is important to mention another significant advantage of this approach. When selecting applications generated for the best chromosome in each generation, the result is a set of very few applications with very good coverage. So for example, for the GA-optimization running 20 generations we can get the best 20 applications from every generation that are able to achieve 98.1 percent coverage. These applications can be used for very efficient regression testing.

The computational burden of the GA optimization was as follows. Evaluating one application in simulation took an average of 12.626 seconds, generating one application around one second and preparing a new population in 0.095 second. Experiments ran on a 3.33 GHz Intel® Core™ i5 CPU with 8 GB of RAM using the Questa simulator.

**SUMMARY**

We have shown that GA easily integrates into a Questa UVM verification environment, is computationally efficient, and significantly reduces the time and effort required to achieve needed coverage for ASIP based designs.

**Figure 3. The comparison of standard verification approaches and the proposed GA-driven approach.**
VERIFICATION ACADEMY
The Most Comprehensive Resource for Verification Training

21 Video Courses Available Covering
• Intelligent Testbench Automation
• Metrics in SoC Verification
• Verification Planning
• Introductory, Basic, and Advanced UVM
• Assertion-Based Verification
• FPGA Verification
• Testbench Acceleration
• Power Aware Verification
• Analog Mixed-Signal Verification

UVM and Coverage Online Methodology Cookbooks
Discussion Forum with more than 5000 topics
UVM Connect and UVM Express Kits

www.verificationacademy.com