Micro-processor verification using a C++11 sequence-based stimulus engine.

DVCON 2017

Stephan Bourduas
(`stephan.bourduas@cavium.com`)

Chris Mikulis
(`chris.mikulis@cavium.com`)

March 1, 2017

Agenda

1. Motivation, Toolchain description.
2. Sequence generator internals (architecture, code examples).
3. Experimental results.
4. Conclusions, Future work.

Introduction

Motivation

UVM has become the de-facto industry standard for design verification.

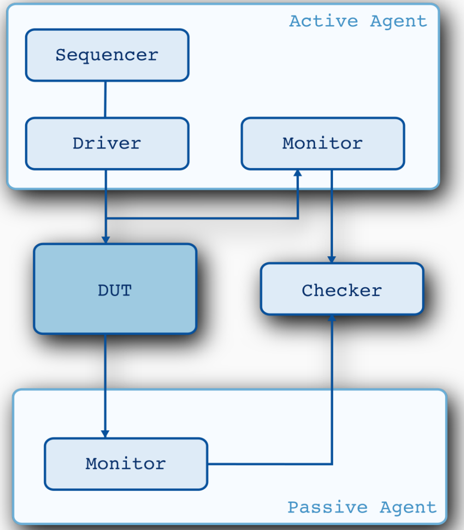
Why didn't we use UVM?

- UVM does not meet our stimulus needs for core verification.
  - We decided to create our own sequence-based assembly generator (SGen) using new features introduced in C++11.

Why C++11?

- Lambda functions enable the creation of closures that capture variables in scope.
- Improved random number generation libraries.
- Polymorphic function wrappers facilitates passing and storing references to callable objects.
- Regular expressions.
- The auto specifier allows the compiler to deduce types.

Stimulus Generation with UVM



The generic UVM testbench:

- Active agent injects stimulus into the DUT.
- Passive agent monitors output.
- Checker correlates inputs and outputs.

Assumes stimulus is generated by a SV seq/driver pair.

- Not the case for core verification!

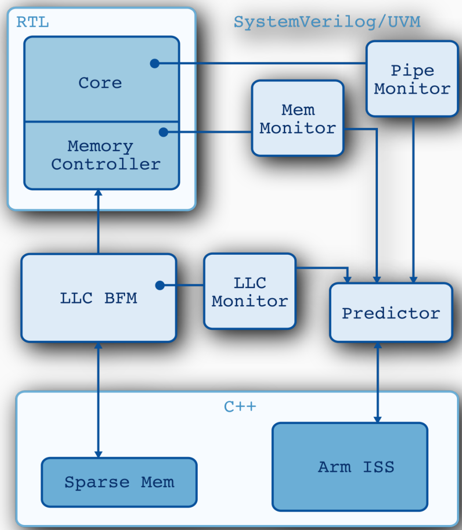
Core Testbench Architecture

Our core testbench:

- Mixed language environment.
- The stimulus is a program binary loaded into memory.
- Core executes program and the BFM reacts.
- UVM seq/drivers used for side-band interfaces.

UVM seq/driver paradigm insufficient:

- We needed to explore alternatives.



Why SGen?

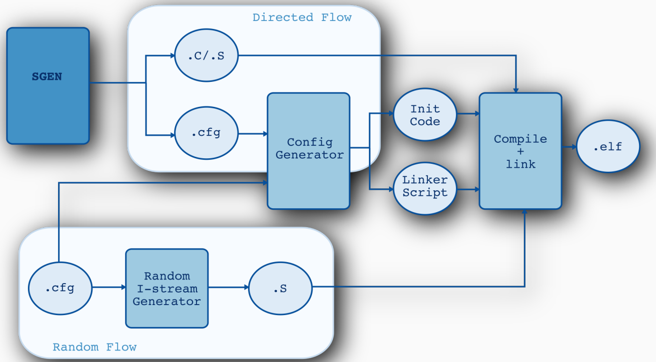
Prior to SGen we had 2 methods to stimulute our design:

1. Hand-written C or assembly tests.
2. Knob-based assembly generator called PPIGen.

The approaches covered opposite ends of the stimulus spectrum:

- Directed tests used for one-off verification tasks and bring up.
- PPIGen is too random... controllability decreases as the number of knobs increases.
- We needed to bridge the gap between directed and fully random stimulus.

Integrating SGen Into the PPIGen Toolchain



PPIGen

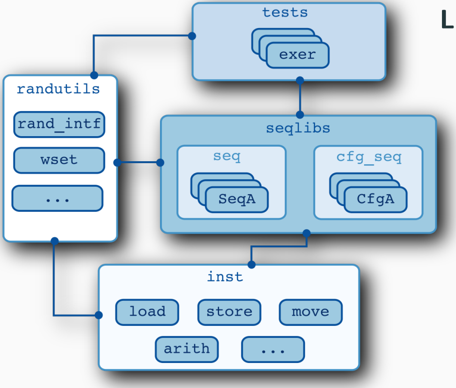
- Directed and random modes.
- Generates init code and linker scripts based on config input (knobs).
- Compiles and link assembly code to generate .elf file.

SGen

- Piggybacks onto the PPIGen directed flow.
- Generates assembly and config files.

Sequence Generator Tool

Software Architecture



Library descriptions:

- inst: hierarchy of instructions organized by type.
- seq: sequences written with specific intent.
  - Includes stimulus and config sequences.
- tests: tests achieve goals by using 1 or more sequences.
- randutils: provides utility classes that enable randomization.

Randomization Without Constraints

Weighted set:

- Parametrizable container that holds items and their associated weights.
- pick method selects a item randomly from the set.
- pick_and_delete method selects and removes an item from the set.
- Can contain weighted sets for recursive picking.

Random interface (base class):

- Provides ability to randomize fields belonging to derived classes.
- Maintains a list of lambda functions that are executed in fifo order when the randomize method is called.
- Lamda functions are added to an object via the push_callback method.
- The push_check method can be used to install callbacks that check for “constraint violations”.

Randomization — The Weighted Set Class

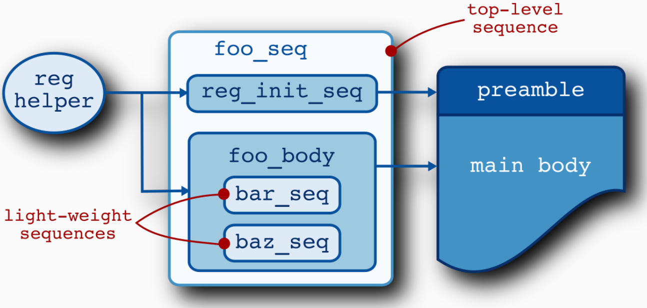
```
// Create 2 weighted sets of chars with 2 items each.
// Note that the syntax uses C++11 list initialization .
wset<char> w1( {{'a', 100}, {'b', 200}} );
wset<char> w2( {{'c', 200}, {'d', 800}} );

// Create a nested weighted set
wset<wset<char>> w;

// Add items to the weighted set
// w2 has a higher weight.
w.add_item(w1, 100);
w.add_item(w2, 900);

// Pick a character from nested wset.
// Template magic will make the call recursive .
char c = w.pick();
```

Randomization — The Random Interface Class

```
// foo.h
class foo : public rand_intf {
    int x;

    foo() {
        push_callback(
            // Set x to 1 by default
            [this]() { x = 1; }
        );

        push_check(
            // make sure x is never
                greater than 10;
            [this]() -> bool { return (x
                <= 10); }
        );
    };
};
```

```
// user_code.cc
// Create a weighted set with 2 items
    with equal weights.
// The set contains an illegal value
    of 20 that will cause the
    simulation to fail if picked.
wset<int> w( {{10, 100}, {20, 100}} );

auto f = new foo();
f->push_callback(
        // Add another callback to
            override the default
        [&](){ f->x = w.pick(); }
    );

// execute all lambda functions in
    fifo order.
f->randomize();
```

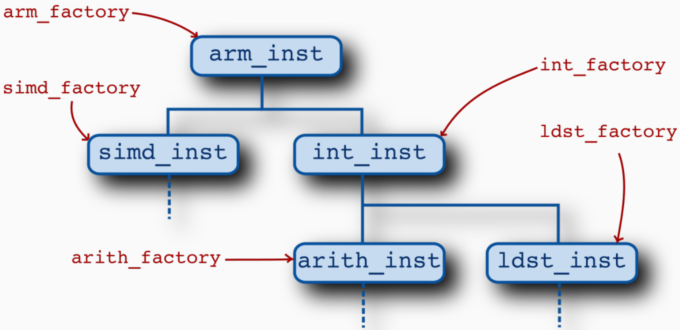
Anatomy of a Sequence



- A sequence run generates a preamble that sets up base, scratch, index and offset registers and the main body.
- The reg_helper class provides random register allocation and reg_init sequence generates the preamble.
- A light-weight sequence is typically short and thus generating a preamble would be wasteful; these sequence rely on the top-level sequence for configuration.

Factories

- A hierarchy of factories reflect the class hierarchy in the `inst` package.
- arm_factory can instantiate any class that derives from arm_inst.
- The get_types method returns array of registered type names:
  - Used to instantiate types belonging to a specific factory
  - The user need not know the names of registered types.
  - Facilitates writing generic sequences (new types are automatically used).



arm_factory

arm_inst

simd_factory

int_factory

simd_inst

int_inst

ldst_factory

arith_factory

arith_inst

ldst_inst

Simple Sequence Example

```
auto rand_w = wset<unsigned>(1,100);  // random number between 1−100
auto& f = simd_factory::instance();   // get reference to desired factory
auto type_vec = f.get_type_instances(); // return vector of type instances

// create weighted set of instances with randomized weights.
wset<simd_factory::inst_type> inst_wset(type_vec
    , [&rand_w](){ return rand_w.pick(); });

// generate random number of instructions.
unsigned num = rand_w.pick();
for(int n = 0; n < num; ++n) {
    auto inst = inst_wset.pick();
    inst−>randomize();
    do_item(∗inst);
};
```

Refer to listing 3 in the paper for a more detailed code example that shows register initialization and instruction randomization.

Experimental Results

Compile and Link Times

Table 1: Incremental compile and link times reported by unix `time` utility.

Modified base class	user time	cpu time
inst	4.5s	1.6s
seq	3.5s	1.3s
test	1.0s	0.5s
all	4.9s	1.8s

- Table 1 shows compile and link times for when files are modified in different packages.
- The inst package has the most reverse dependencies.
- The table shows compile and link times for when files are modified in each package.
- The worst case compile time was 5s.

NOTE: Full compile of our testbench took 9m and incremental compile after touching a single SV test took 6m.

Regressing RTL Changes

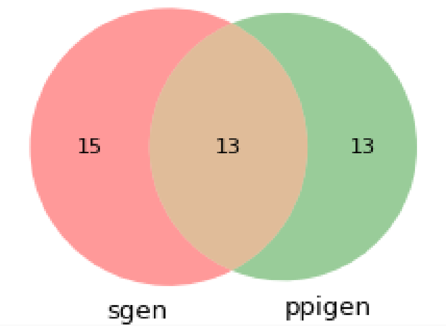
Table 2: Failures broken down by test type for recent regression and exerciser runs.

		Failures		
Series	Tests	Directed	PPIGen	SGen
reg0	421	2	0	6
reg1	446	1	0	15
reg2	417	2	0	29
reg3	410	21	16	34
reg4	388	1	1	8
exer0	47	–	0	5
exer1	46	–	0	6
exer2	41	–	9	26
exer3	35	–	1	0
exer4	55	–	1	0

- RTL designer has added SGen exers to their regression flow.
    - reg runs consist of directed tests as well as PPIGen and SGen exers.
    - exer runs consist of only random exercisers.
- Directed failures do not necessarily gate check-in.
- SGen has consistently caught bugs that would have escaped standard regression.

PPIGen and SGen Exerciser Runs

Figure 7: Overlap of failure buckets for SGen and PPIGen



- 25k exerciser runs per generator were run over a 1 month period.
- 1731 total failures:
  - SGen accounted for 1560
  - PPIGen accounted for 171
- Once we bin the failures we see that both tools are doing a good job:
  - 41 total buckets with only 13 overlapping.
- SGen is more efficient at hitting bugs despite PPIGen being the more mature tool.

Runtime Speed

Primary motivation for C++11 was speed:

- Typical exerciser run generates tests with 25k–50k instructions.
- For 500 runs:
  - the average execution time of SGen was 709ms.
  - The number of instructions generated per second was 31k.
- The tool adds no overhead (computes and licenses) to our simulation times.

Conclusion and Future Work

Conclusion

- We bridged the gap between directed and fully random stimulus by creating a sequence-based generator using C++11.
- We were able to express complex dependencies between random variables despite the lack of a constraint engine.
    - Weighted set, random interface and lambda functions.
- New features introduced in C++11 were key enablers.
    - Fast compile and run times increased productivity.
- SGen is currently being used in production to verify the Cavium ThunderX2 core
    - Results show that it is better at uncovering certain types of errors than existing tools.
    - Generated more failures than PPIGen for the same number of runs.

Future Work

- Continue adding support for SIMD and FP instructions.
- Continue adding to sequence library.
- Improve configuration randomization.
- Explore possibility of using an interpreted language for test writing (i.e. Python front end).

End.