HDL verification with Maia

Maia is specifically designed to automate the test and specification of HDL designs.

- C-like: *no* knowledge of VHDL, Verilog, or 'verification' required
- *No* low level coding (clocks, resets, signals, delays, races, etc)
- Tests can be written by software engineers, tech authors, etc, *but*…
- Tests will normally be written by the designers themselves in a **Test Driven Development** (TDD) flow
- The test unit is an individual module: leaf or hierarchical
- The compiler generates a *self-checking* testbench, with error reporting
- You still need a simulator!
- The compiler driver automatically runs a batch-mode simulation of the TB and your DUT (dual-language simulator required for VHDL designs)
- Free (beer) compiler (C++, 9-pass, Linux/Windows)

Trivial example: pipelined MAC

```
#define PL 3                                  // DUT has 3-level pipe
DUT {
   module MAC1
      @(.stages(PL))
       (input  RST, CLK,
        input  [3:0] A, B,
        output [9:0] Q);
   [RST, CLK, A, B] -> [Q];                   // drive declaration
   create_clock CLK;                          // default timing, rising edge
}
main() {
   var4  ina = 0, inb = 0;                    // 4-bit, 4-state
   var10 sum = 0;                             // 10-bit, 4-state
   [1, .C, -, -] ->PL [0];                    // confirm resets Ok (A,B inputs D/C)
   for(int i=0; i<16; i++, ina++)
      for(int j=0; j<16; j++, inb++) {
         sum += ina *$8 inb;                  // 8-bit multiplier, 4-bit inputs
         [0, .C, ina, inb] ->PL [sum];        // output is checked after PL cycles
      }
}
```

DUT definition: Verilog-style module Declaration (module can be VHDL), with sdc-style clocks, virtual clocks, timing constraints

control language: C-like, with extensions

Simulation output (assuming correctly-code HDL!)

```
(Log)          (2590 ns) 257 vectors executed (257 passes, 0 fails)
```

Maia: history (1)

Maia started life (in 1999) as a VHDL library which read test vectors from a file and applied them to a DUT. Each line of the file contained a set of inputs, and a set of expected outputs following a clock edge:

```
[0, 1, abcd, 2345] -> [bcef0123, 45] # I/Os are hex constants
```

Sampled outputs were tested against the expected values. This was enough to create simple self-checking tests, but had major limitations:

- Sequential execution with no branching/decision making ⇒

  not responsive, no ability to modify test depending on DUT state (drive different input values depending on a control output, etc)
- Can only drive and test against constant values ⇒

  everything must be pre-computed; all inputs and expected outputs must be known in advance
- No variables, expressions, functions, etc ⇒

  no algorithmic testing or independent computation of ins/outs

Maia: history (2)

There were many other issues with the VHDL test library:

- Required some VHDL coding to specify clocks, signal names, widths, etc
- Difficult to test pipelines; must code expected values later in the output sequence, and explicitly add vectors for pipe setup/flush
- Fixed timing – only suitable for delta-delay sims
- All clocked: no combinatorial output testing
- No ability to wait for specific conditions on the DUT
- One linear flow makes the test effectively single-threaded

In principle, most of these issues can be handled with ad-hoc extensions in the VHDL code, but this would be fragile. The solution was instead to define a test language and write a compiler for it:

- Before: fixed testbench reads fixed test vectors from a file
- After: source code contains test spec; compiler generates testbench
- Derived from a language and compiler which were part of a SMART-funded automated processor-verification system
- First internal release 2009; public 2019.11 release at maia-eda.net

Maia: program structure (1)

- A *DUT section* is used to declare the DUT and the format of any test vectors (***drive declarations***), and to declare clocks/internal signals/etc
- The test code is a collection of C-like functions (with entry at `main`): functions don't need to be declared, and can be entered in any order
- Hardware is tested with ***drive statements*** (or direct driving and reading of HDL signals)
- Very simple test programs don't require a `main` function – can just enter a list of drive statements

Drive declaration

- One or more *drive declarations* in the DUT section specify inputs to drive, and the outputs to test after an appropriate delay
- If an input also appears in a `create_clock` statement then the drive declaration is clocked/sequential; otherwise it is combinatorial

```
[rst156, clk156, ready, valid, din] -> [dout, full];
```

- I/Os may be DUT inputs/outputs, or declared internal signals (currently Verilog only): internal inputs are automatically preloaded (force/release)

Maia:program structure (2)

Drive statements

- Similar to the original version, but:

Special input drives:
.C clock
.R release/etc

Optional pipe Level: `->n`

Inputs and outputs can be arbitrary expressions

```
[0, .C, 2*x, y, z] -> [foo(), 0x64, 2*out2()];
```

- Clocked drives (containing a `.C`) drive a specified `create_clock` waveform on an input *and advance time*
- The signals are defined in the matching drive declaration
- Clocked drives advance time to the next *operating point*: other statements (calculations, function calls, etc) then execute in zero time, until the next drive (or wait) statement
- Incorrect outputs are logged, or stop the sim, as required
- Clocked drives set up pipelined checkers for all required tests, and default to a test after the next clock edge

Maia: simulation time

- Time is advanced by *wait statements* and *drive statements*
- A clocked drive advances time between *operating points (*OPs*)*
- An OP is (conceptually) the time at which clocked outputs are stable, but before any inputs need to be driven to set up for the next clock
- You can execute any other Maia statements in between drive statements: these occur in zero simulation time, at the OP
- For testing combinatorial circuits the compiler chooses an OP to step through combinatorial drive statements
- Timing (clock waveforms, setup, hold, output delay) can be specified with Primetime-compatible syntax in the DUT section
- Default (unit delay) sims have a 10ns clock, with an operating point a little before the rising edge (eases waveform viewing)
- A large number of unit tests confirm that the compiler Verilog output is race-free and behaves as expected on a range of simulators

Maia: control language (1)

- Simplified C look-and-feel: no pointers, no standard library
- Control statements are generally identical to C
- Full built-in C preprocessor
- Minor changes and additions: multi-level break, `for all`, simple file and console I/O in language, default initialisation, left-to-right evaluation
- Major changes: drive statements, type system, references, trigger functions, thread functions, bitslices

Type system (1)

- Both static and dynamic type checking, with configurable strength (level 0 is script-like, 1 is C-like (default), 2 is strongly typed)
- 2-state (`01`) and 4-state (`01xz`) integers with arbitrary sizes (bigint arithmetic): `bit256` is 2-state 256 bits, `var102` is 4-state 102 bits, etc
- Both C and Verilog-style constants: `0xabcd`, `1.4E9`, `4'hx`, `5'd4`, etc
- `int`, `bit`, `var`, `bool`, `kmap`, `stream`, `array`, `struct` types
- Unconstrained types: `ubit`, `uvar` (size not known in advance; useful for generic functions)

Maia: control language (2)

Type system (2)

- Data objects have no properties apart from their size: they are *not* signed, unsigned, integer, floating-point, etc. They are *just data*
- Complexity is instead provided by *operators*:

`-`	implicitly-sized unsigned integer subtraction
`-#`	implicitly-sized signed (2's complement) integer subtraction
`-#$21`	21-bit signed integer subtraction
`.R<<`	Rotate left
`.F2*`	Double-precision floating-point multiply
`myvar.(x:y)`	Bitslice

- Models hardware design: memory locations just connect to function units
- This is the exact *opposite* of object orientation!
- An `int` type gives traditional signed behaviour for software convenience (loop indexing, etc)

Maia: control language (3)

Some C code can be copied direct. This is valid C and Maia:

```
#ifdef __MAIA__
#define printf report   // Maia has a printf-style 'report' statement
#else
#include <stdio.h>      // C-only
#include <assert.h>
#endif

struct s1 {
   int x, y;
} a = {40+2, 42*2-42+1};

main() {
   assert(a.x == 42 && a.y == 43);
   printf("a.x: %d; a.y: %d\n", a.x, a.y);
}
```

Maia: streams

Simple file I/O is built into the language: optimised for reading sets of inputs and expected outputs

```
//! Apply the variable plaintext known answer data, and check the results
run_vpkat() {
   stream {    // stream declaration opens, reads, and checks the named file
      mode 1;
      file "vectors/des_vpkat.dat";
      format "%i %64'h %64'h", round, plain, cipher;
   } vpkat;
   ...
   for all vpkat
      passcount += drive(mode, vpkat.round, key, vpkat.plain, vpkat.cipher);
   }
}  // run_vpkat()
```

Input file:

```
0  8000000000000000 95F8A5E5DD31D900
1  4000000000000000 DD7F121CA5015619
...etc
```

Maia: summary

- Automated unit testing for VHDL and Verilog designs
- Download from http://www.maia-eda.net
- Compiler, LRM, tutorial, FAQs, resources, scripts
- You need a Verilog-only simulator for Verilog DUTs (Icarus v10 is free and works); or a mixed-language simulator for VHDL DUTs (some vendors supply free dual-language simulators)
- The *mtv* compiler compiles Maia sources to Verilog. The *rtv* driver runs an entire batch-mode simulation (with Maia, VHDL, and Verilog sources), using a configuration file which describes vendor simulators