Options for Testing Chisel Designs

- testing using assert and printf in Chisel
- testing Verilog using VCS
- testing from within Scala
  - test C++ executable
  - test Verilog using VCS
- testing inside C++ simulator
  - VCD debugging
  - manual testing from within C++
  - testing using Verilog
Simulation Options

<table>
<thead>
<tr>
<th>Simulated Cycles</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chisel C++ (gcc −O0)</td>
<td></td>
</tr>
<tr>
<td>Chisel C++ (gcc −O3)</td>
<td></td>
</tr>
<tr>
<td>Chisel Verilog (VCS)</td>
<td></td>
</tr>
<tr>
<td>Chisel Verilog (Virtex−6)</td>
<td></td>
</tr>
</tbody>
</table>
Chisel Workflow

```
Mux2.scala ➔ scala compiler ➔ bytecodes ➔ jvm chisel builder ➔ Mux2.v

net list + power, area, and speed estimates ➔ dc ➔ Mux2.v

jvm verilog backend ➔ vcs ➔ verification

jvm cpp backend ➔ exec ➔ Mux2

Mux2.v ➔ g++ ➔ Mux2
```

```
Mux2.scala bytecodes
Mux2.cpp
scala
compiler
jvm
chisel
builder
jvm cpp backend
Mux2 g++
+ 1 2
Mux2.v jvm verilog backend
verification vcs
simulation
exec
net list +
power, area, and speed estimates
dc
```
during simulation

- printf prints the formatted string to the console on rising clock edges
- sprintf returns the formatted string as a bit vector

format specifiers are

- %b – binary number
- %d – decimal number
- %x – hexadecimal number
- %s – string consisting of a sequence of 8-bit extended ASCII chars
- %% – specifies a literal

the following prints the line "0x4142 16706 AB" on cycles when \( c \) is true:

```hs
val x = Bits(0x4142)
val s1 = sprintf("%x %s", x, x);
when (c) { printf("%d %s\n", x, s1); }
```
- simulation time assertions are provided by `assert` construct
- if `assert` arguments are false on rising edge then
  - an error is printed and
  - simulation terminates

The following will terminate after 10 clock cycles:

```scala
val x = Reg(init = UInt(0, 4))
x := x + UInt(1)
assert(x < UInt(10))
```
- produce Verilog from Chisel
- write tests in Verilog harness
- use waveform debugger
tests written in Chisel

Chisel
- compiles,
- runs, and
- talks to DUT using pipes

User
- sets inputs + get outputs using
  - Chisel data to get nodes and tables from nodes to values
- specifies nodes to trace
Chisel Based Testing Details

```
package Tutorial
import Chisel._
import scala.collection.mutable.HashMap
import scala.util.Random

class Combinational extends Module {
  val io = new Bundle {
    val x = UInt(INPUT, 16)
    val y = UInt(INPUT, 16)
    val z = UInt(OUTPUT, 16)
  }
  io.z := io.x + io.y
}

class CombinationalTests(c: Combinational)
  extends Tester(c, Array(c.io)) {
  defTests {
    var allGood = true
    var vars = new HashMap[Node, Node]()
    val rnd = new Random()
    val maxInt = 1 << 16
    for (i <- 0 until 10) {
      vars.clear()
      val x = rnd.nextInt(maxInt)
      val y = rnd.nextInt(maxInt)
      vars(c.io.x) = UInt(x)
      vars(c.io.y) = UInt(y)
      vars(c.io.z) = UInt((x + y) & (maxInt - 1))
      allGood = step(vars) && allGood
    }
    allGood
  }
}
```

- user subclasses Tester defining DUT and testNodes and tests in defTests body
- vars is mapping from testNodes to literals, called bindings
- step runs test with given bindings, where var values for input ports are sent to DUT, DUT computes next outputs, and DUT sends next outputs to Chisel
- finally step compares received values against var values and returns false if any comparisons fail
object chiselMainTest {
    def apply[T <: Module]
        (args: Array[String], comp: () => T)(tester: T => Tester[T]): T
}

and used as follows:

chiselMainTest(args ++ Array("--compile", "--test", "--genHarness"),
        () => new Combinational(){
            c => new CombinationalTests(c)
        }
ChiselMain* Arguments

--targetDir target pathname prefix
--genHarness generate harness file for C++
--backend v generate verilog
--backend c generate C++ (default)
--compile compiles generated C++
--test generates C++ with test plumbing
--vcd enable vcd dumping
--debug put all wires in C++ class file
Running Tests Examples

sbt "project tutorial" "run Combinational ... --compile --test --genHarness"

... 
PASSED

or through makefile

cd CHISEL/tutorial/emulator
make combinational

... 
PASSED
class GCDTests(c: GCD) extends Tester(c, Array(c.io)) {
    defTests {
        val (a, b, z) = (64, 48, 16)
        val svars = new HashMap[Node, Node]()
        val ovars = new HashMap[Node, Node]()
        var t = 0
        do {
            svars(c.io.a) = UInt(a)
            svars(c.io.b) = UInt(b)
            step(svars, ovars)
            t += 1
        } while (t <= 1 || ovars(c.io.v).litValue() == 0)
        ovars(c.io.z).litValue() == z
    }
}
class AdvTester[+T <: Module](val dut: T, val ios: Array[Node])
  extends Tester[T](dut, ios) {

  val preprocessors = new ArrayBuffer[Processable]()

  val postprocessors = new ArrayBuffer[Processable]()

  // like step but does pre and post processing and work
  def takestep(work: => Unit = {}) = ...

  def until(pred: => Boolean, maxCycles: Int = ...)(work: => Unit): Boolean = ...

  def do_until(pred: => Boolean, maxCycles: Int = ...)(work: => Unit): Boolean = ...

  def eventually(pred: => Boolean, maxCycles: Int = ...)
  =
    until(pred, maxCycles){ }

}
DecoupledSource
- queue from tester to DUT
- tester enqueues data onto queue
- handlers moving data from queue to decoupled interface
- can use `until` to wait until data shows up on queue

DecoupledSink
- queue from DUT to tester
- tester sees data on queue
- handlers moving data from decoupled interface to queue
- can use `until` to wait until data shows up on queue
Imagine testing your coprocessor with AdvTester (instead of rocket core):

```scala
val commands = new DecoupledSource(dut.io.cmd, ...)
val responses = new DecoupledSink(dut.io.resp, ...)
...
defTests {
    ...
    commands.inputs.enqueue(TestCmd.setup(0, 1, 10))
    until(!responses.outputs.isEmpty) {
    val resp = response.outputs.dequeue()
    assert(resp.data == ..., "test 1 failed: bad response")
    assert(resp.rd == 10, "test 1 failed: bad rd returned")
    ...
}
```

- memory can also be handled in tester using Scala queues and Scala memory and a process that gets stepped to handle mem requests.
- Cycle accurate simulator
  - Easy way to debug designs
- Compile Chisel to one C++ class
  - Topologically sorts nodes based on dependencies
- Simulates using two phases
  - clock_lo for combinational
  - clock_hi for state updates
- Using fast multiword C++ template library
  - Now though expand in Chisel backend
  - Use same representation
In order to construct a circuit, the user calls `chiselMain` from their top level `main` function:

```java
object chiselMain {
    def apply[T <: Module]
        (args: Array[String], comp: () => T): T
}
```

which when run creates C++ files named `module_name.cpp` and `module_name.h` in the directory specified with `-targetDir dir_name` argument.

`chiselMain(Array("--backend", "c", "--targetDir", "/../emulator"),
    () => new GCD())`
template <int w>
class dat_t {
  public:
    const static int n_words = ((w - 1) / 64) + 1;
    val_t values[n_words];
    inline val_t lo_word ( void ) { return values[0]; } 
    ...
}

template <int w> dat_t<w> DAT(val_t value);
template <int w> dat_t<w> LIT(val_t value);

template <int w> std::string dat_to_str (dat_t<w> val);

std::string read_tok(FILE* f);

template <int w> void str_to_dat(std::string str, dat_t<w>& res);
class mod_t {
public:
    std::vector< mod_t* > children;
    virtual void init ( void ) { };
    virtual void clock_lo ( dat_t<1> reset ) { };
    virtual void clock_hi ( dat_t<1> reset ) { };
    virtual void print ( FILE* f ) { };
    virtual bool scan ( FILE* f ) { return true; };
    virtual void dump ( FILE* f, int t ) { };
};
GCD.h – the header for the single class
GCD.cpp – the implementation of the single class
GCD-emulator.cpp – the harness which cycles the design
GCD.vcd – produced when running design with vcd output
```cpp
#include "emulator.h"

class GCD_t : public mod_t {
    public:
        dat_t<1> GCD__io_v;
        dat_t<16> GCD__io_b;
        dat_t<1> GCD__io_e;
        dat_t<16> GCD__y;
        dat_t<16> GCD__y_shadow;
        dat_t<16> GCD__io_a;
        dat_t<16> GCD__x;
        dat_t<16> GCD__x_shadow;
        dat_t<16> GCD__io_z;

        void init ( bool rand_init = false );
        void clock_lo ( dat_t<1> reset );
        void clock_hi ( dat_t<1> reset );
        void print ( FILE* f );
        bool scan ( FILE* f );
        void dump ( FILE* f, int t );
};
```
Name Mangling Scheme

- chisel object names are mangled to
  - maintain uniqueness and avoid name conflicts
  - maintain hierarchical membership
  - avoid problems with C++ naming convention

- basic scheme is pathname consisting of
  - Module name first followed by __
  - hierarchy elements separated with _’s in order with
    - numbers for vector elements
    - names for bundle fields
  - actual object name last

- examples
  - val io = Bundle{ val x = UInt(width = 32) } produces
    - A__io_x
  - ... Vec.fill(2){ Decoupled( Bool() ) } produces
    - B__io_ports_0_ready
```cpp
#include "GCD.h"

void GCD_t::init ( bool rand_init ) {
    { GCD__y.values[0] = rand_init ? rand_val() & 65535 : 0; }
    { GCD__x.values[0] = rand_init ? rand_val() & 65535 : 0; }
}

void GCD_t::clock_lo ( dat_t<1> reset ) {
    val_t T0__w0;
    ...
};

void GCD_t::clock_hi ( dat_t<1> reset ) {
    GCD__y = GCD__y_shadow;
    GCD__x = GCD__x_shadow;
}

void GCD_t::print ( FILE* f ) {
    fprintf(f, "%s", TO_CSTR(GCD__io_z));
    fprintf(f, "\t", TO_CSTR(GCD__io_v));
    fprintf(f, "\n");
    fflush(f);
}

bool GCD_t::scan ( FILE* f ) {
    str_to_dat(read_tok(f), GCD__io_a);
    str_to_dat(read_tok(f), GCD__io_b);
    str_to_dat(read_tok(f), GCD__io_e);
    return(!feof(f));
}

void GCD_t::dump(FILE *f, int t) {
}
```
#include "GCD.h"

int main (int argc, char* argv[]) {
    GCD_t* c = new GCD_t();
    int lim = (argc > 1) ? atoi(argv[1]) : -1;
    c->init();
    for (int i = 0; i < 5; i++) {
        dat_t<1> reset = LIT<1>(1);
        c->clock_lo(reset);
        c->clock_hi(reset);
    }
    for (int t = 0; lim < 0 || t < lim; t++) {
        dat_t<1> reset = LIT<1>(0);
        if (!c->scan(stdin)) break;
        c->clock_lo(reset);
        c->print(stdout);
        c->clock_hi(reset);
    }
}

use `-vcd arg` to have simulation produce VCD output

run your compiled C++ emulation app for a number of cycles
  - specifying the number of cycles as a first argument

can view waveforms with
  - `vcs` – commercial
  - GTKWave – open source

can hierarchically focus on particular signals

can view in a variety of formats
- test Chisel code by manually
  - setting circuit inputs directly in your C++ code
  - inserting `printf`'s in your C++ code

- in your C++ harness insert calls to
  - `str_to_dat(read_tok(f), GCD__io_a)` to set values
  - `TO_CSTR(GCD__io_z)` to create string for printing

- in your chisel code
  - wrap nodes with `debug` as in `debug(io.z)`

- in your `chiselMain`
  - you can add `--debug` arg to get everything available in object
for simple tests can write test vector files
check out Chisel tutorial code