Chisel @ CS250 – Part III – Lecture 11

Jonathan Bachrach
EECS UC Berkeley

September 28, 2012
Options for Testing Chisel Designs

- 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++
Simulation Options

The image shows a graph with a logarithmic scale for both the x-axis and the y-axis. The x-axis represents the number of simulated cycles, ranging from $10^5$ to $10^9$. The y-axis represents time in seconds, ranging from $10^0$ to $10^6$.

The graph includes several lines, each representing a different simulation option:

- **Chisel C++ (gcc -O0)**: A solid black line.
- **Chisel C++ (gcc -O3)**: A dashed black line.
- **Chisel Verilog (VCS)**: A dotted gray line.
- **Chisel Verilog (Virtex-6)**: A dot-dashed gray line.

The lines show the relationship between the number of simulated cycles and the time taken for each simulation option.
Chisel Workflow
- 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
```scala
package Tutorial
import Chisel._
import scala.collection.mutable.HashMap
import scala.util.Random

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

class CombinationalTests(c: Combinational)
extends Tester(c, Array(c.io)) {
  defTests {
    var allGood = true
    val 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) = UFix(x)
      vars(c.io.y) = UFix(y)
      vars(c.io.z) = UFix((x + y)&(maxInt-1))
      allGood = step(vars) && allGood
    }
    allGood
  }
}

class Tester[T <: Component]
(val c: T, val testNodes: Array[Node])
  def defTests(body: => Boolean)
  def step(vars: HashMap[Node, Node]): Boolean

- 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 <: Component]
        (args: Array[String], comp: () => T)(tester: T => Tester[T]): T
    }

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

-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) = UFix(a)
      svars(c.io.b) = UFix(b)
      step(svars, ovars)
      t += 1
    } while (t <= 1 || ovars(c.io.v).litValue() == 0)
    ovars(c.io.z).litValue() == z
  }
}
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:

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

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

```scala
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 ) { };
};
C++ Simulator Outputs

- 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
#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 );
};
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
- Component 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 = Bits(width = 32) } produces
  - A__io_x
- ... Vec(2){ FIFOIO(){ 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, "%s", " ");
    fprintf(f, "%s", 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_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
  - `T0_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