A Short Tutorial on the `circuit` and `statereg` Classes

Matthew I. Frank
Dept. of Electrical and Computer Engineering
University of Illinois, Urbana-Champaign
mif@uiuc.edu

September 17, 2003

The `circuit` and `statereg` classes (defined in `circuit.h`), are designed to provide a programming model that will be familiar to `verilog` or `vhdl` designers. The basic design philosophy is that of designing Moore style finite state machines, where there is some set of state elements (represented by the `statereg`s) that changes on clock pulses (*e.g.*, negative edges), and some combinational logic for the transition function (represented by `circuits`) that is evaluated between clock pulses.

The best way to understand `circuit`s and `statereg`s is by example. Suppose we wish to implement a 16 bit counter that also tells us what its output was *last* cycle. An implementation of this state machine is shown in Figure 1.

This counter demonstrates most of the features of the `circuit` and `statereg` classes. Note that `statereg`s are declared much like any other variable in C++: `statereg<`*`typename`*`>` *`varname`*. [1] `statereg` assignments all occur simultaneously at clock pulses. The value set in a `statereg` at the *previous* clock pulse can be read by using the parenthesis notation (*e.g.*, `count()`). Note that `statereg`s can hold arbitrary C++ types. They are not limited to holding only integers.

The transition function for the `circuit` is held inside a private function called `recalc`. The recalc function defines the transition behavior of the circuit. In this case `count` is changed conditionally depending on the `change` and `advance` signals. The `if` statements thus work much like multiplexors. Note that if neither `advance` nor `change` are true then `count` will be set to the same value at the next clock pulse that it held at the previous clock pulse. As with `statereg`s that can hold arbitrary C++ types, the `recalc` function can hold arbitrarily complicated C++ code. Thus if your transition function is very complicated, you should consider defining helper functions and classes to make your code cleaner and easier to understand. (Consider, for example, the recalc function in `decode_circuit.h` that calls out to additional C++ objects to actually decode each instruction.)

`circuits` can be *composed*. This is done by using the `attach()` primitive on `inports`. An example of circuit composition is shown in Figure2. Here we define a second circuit that sets its output state true or false depending on whether or not its input is equal to 17 or not. Then, inside the `main()` function, we attach together our counter with our new counter that is more interesting than our original counter.

The `main()` function also demonstrates how `circuit`s and `statereg`s are invoked. The function contains a `while` loop that alternately `pulse`s all the `statereg`s and then calls `recalc` on all the `circuits`. You will see a similar `main` function inside the pipelined simulator.

There is one additional useful feature included in `circuit.h`, which is intended mainly for clock gating an output register. This is the `member_outport` class that can be used to expose a combinational function on the inputs to a circuit to other circuits. An example of the use of `member_outport` can be

[1] The type `uint16_t` is the standard POSIX portable typename for an unsigned 16 bit integer. Thus you can use `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`, for unsigned integers and `int8_t`, `int16_t`, `int32_t`, `int64_t`, for signed integers of the appropriate size. These types should be defined in `<inttypes.h>` on any POSIX conforming UNIX system.

```
class counter : circuit {
  // This is the transition function:
  void recalc() {
    if (advance) {
      count = count() + 1;
    }
    if (change) {
      count = setval();
    }
    prev_count = count();  // statereg assignments occur concurrently!!!
  }
public:
  counter() : setval(), change(), advance(), count(0), prev_count(0) { }
  // inputs
  inport<uint16_t> setval;
  inport<bool> change;
  inport<bool> advance;

  // outputs
  statereg<uint16_t> count;
  statereg<uint16_t> prev_count;
};
```

Figure 1: A 16 bit counter. Note that `statereg` values set on the *previous* clock pulse are gotten by refering the the register with parenthesis after its name (*e.g.*, `count()`). `statereg` assignments that are to take place on the *next* clock pulse all occur concurrently (simultaneously). Thus, the statement `prev_count = count()` *is* correct.

seen in the `scoreboard` circuit, which needs to stall the `fetch` and `decode` stages when it recognizes a hazard.

```
class test17 : circuit {
  void recalc() {
    was17 = false;
    if (value() == 17) {
      was17 = true;
    }
  }
public:
  counter() : value(), was17(false) { }
  // inputs
  inport<uint16_t> value;

  // outputs
  statereg<bool> was17;
};

int main(int argc, char* argv[]) {
  statereg<uint16_t> always_0;
  statereg<bool> always_false;
  counter the_counter;
  test17 the_comparator;

  // wire up the inputs to the counter
  the_counter.setval.attach(&always_0);
  the_counter.change.attach(&the_comparator.was17);
  the_counter.advance.attach(&always_false);

  // wire up the input to the comparator
  the_comparator.value.attach(&the_counter.count);

  while (true) {
    clocked_net::pulse(); // Toggles statereg clocks to change values like
                          // a register in real circuits.
    circuit::level();     // Evaluates recalc() transition functions.
  }
}
```

Figure 2: Circuits can be composed by using the `attach()` primitive on `inport`s. Here we define a second circuit that sets its output register true exactly when its input is equal to the value 17. Then in the `main()` function we declare two circuits (a `counter` and a `test17`) and wire them together using the attach primitive. The `counter`'s `change` input is attached to the comparator's output. The comparator's input is attached to the `counter` output. What does this circuit do?