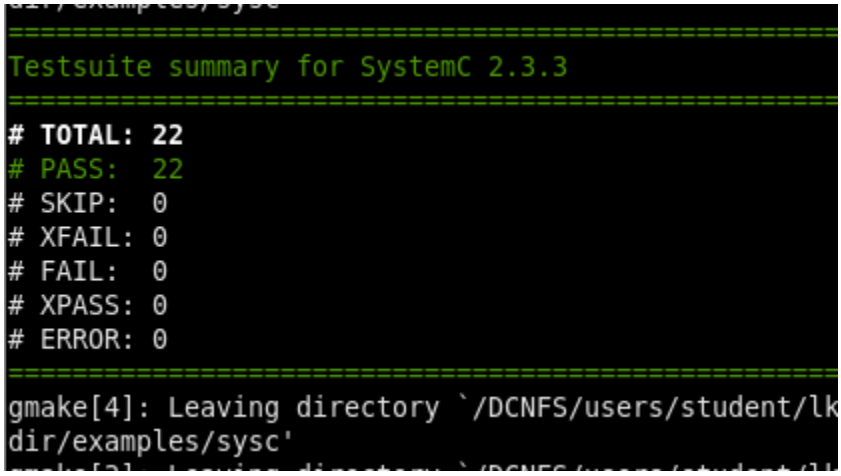
ELEN 503 HW 2 Report Laurence Kim Submission

Part 1: Install SystemC



Part 2:

This portion of the code was fairly straightforward as we had to simply connect the stimulus and the adder by adding a couple lines of code that actually enables this process. Afterwards, I tinkered around with the do it function as to personally see how what tweaks could affect the waveform graph. Ultimately, after doing so, I know that the main difference between sc_signal and sc_buffer is that sc_buffer generates an event at every write(), while sc_signal generates an event only when the value is changed.

```
#include "systemc.h"
SC_MODULE(generator) {
  sc_out<short> sig;
  sc_out<short> buf;

  void do_it(void) {
    wait(5, SC_NS); //5 5,5
    sig.write(5);
    buf.write(5);
    short value = 10;
    wait(10, SC_NS); //15 0,0

    // Modifying the value through the signal
    sig.write(5); // Update the value in the signal
    buf.write(5); // Update the value in the buffer
    wait(10, SC_NS);
```

```
    // Additional code to demonstrate the difference between sc_signal and
sc_buffer
    // Modifying the value through the signal
    sig.write(value); // Update the value in the signal
    buf.write(value); // Update the value in the buffer
    wait(10, SC_NS);


    // Modifying the value through the buffer
    buf.write(value); // Update the value in the buffer
    value = 20; // Modify the local variable value
    wait(10, SC_NS);

    // Output the final values of the signal and buffer
    cout << "Final value of sig: " << sig.read() << endl;
    cout << "Final value of buf: " << buf.read() << endl;

    sc_stop(); // Stop simulation
  }

  SC_CTOR(generator) {
    SC_THREAD(do_it);
    sig.initialize(0);
    buf.initialize(0);
  }
};

SC_MODULE(receiver) {
  sc_in<short> iport;

  void do_it(void) {
    cout << sc_time_stamp() << ":" << name() << " got " << iport.read() <<
endl;
  }

  SC_CTOR(receiver) {
    SC_METHOD(do_it);
    sensitive << iport;
    dont_initialize();
```

```
  }
};

int sc_main(int argc, char *argv[]) {
  sc_signal<short> sig;
  sc_buffer<short> buf;

  generator GEN("GEN");
  GEN.sig(sig);
  GEN.buf(buf);

  receiver REV_SIG("REV_SIG");
  REV_SIG.iport(sig);
  receiver REV_BUF_A("REV_BUF");
  REV_BUF_A.iport(buf);

  // trace file creation
  sc_trace_file *tf = sc_create_vcd_trace_file("wave");
  sc_trace(tf, sig, "sig");
  sc_trace(tf, buf, "buf");

  sc_start();

  sc_close_vcd_trace_file(tf);
  return (0);
}
```

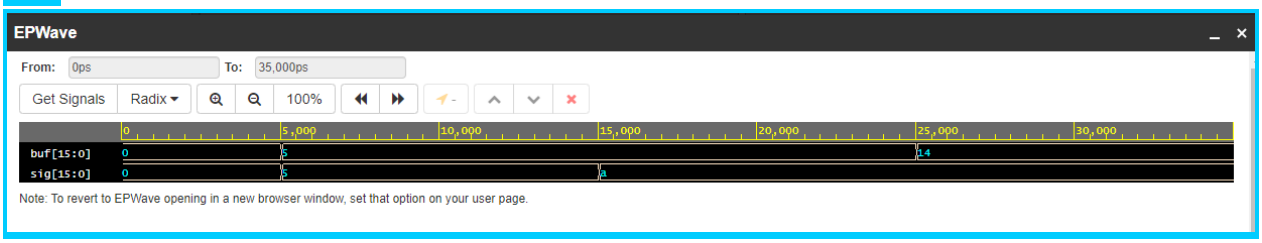
```
[2023-05-26 17:48:43 EDT] g++ -o sim -lsystemc *.cpp  && echo "Compile done. Starting run..." && ./sim
Compile done. Starting run...

        SystemC 2.3.3-Accellera --- May 23 2020 14:33:56
        Copyright (c) 1996-2018 by all Contributors,
        ALL RIGHTS RESERVED
0 s:REV_BUF got 0

Info: (I702) default timescale unit used for tracing: 1 ps (wave.vcd)
5 ns:REV_SIG got 5
5 ns:REV_BUF got 5
15 ns:REV_SIG got 10
25 ns:REV_BUF got 20
Final value of sig: 10
```

```
Final value of buf: 20

Info: /OSCI/SystemC: Simulation stopped by user.
Finding VCD file...
./wave.vcd
[2023-05-26 17:48:44 EDT] Opening EPWave...
Done
```



Part 3:

Fibonacci Sequence

Let me first start by explaining this code by segmenting it into six parts. The first part are abstract interfaces that allow us to actually do the writes and reads to and from the FIFO as well as a reset and num_available method for quality of life implementation. The second part is the actual fifo implementation which has its own class as it builds upon the current sc channel that exists. This class holds a n array of data that has a num_elements and the first to keep track of the numbers in the fifo and the index of the first element within the fifo. The write method is blocking if it is full and lets the producers know likewise. The write_events and read_events are both meant to synchronize with the producer and consumers. The fibonacci generator is the actual producer and simply queues each order of the sequence into the fifo by using a write method. The consumer also extends the sc module like the producer and reads from the fifo. The top module constructs the interconnect of the producer, consumer, and fifo by the names fifo, fibonacci_gen, and consumer respectively. All these instances are connected by sc_ports. Finally we have a main function that runs the simulation with sc_start with the general structure and flow that we have seen in part 1 and the other auxiliary modules that we have referenced thus far.

```
// Code your testbench here
// or browse Examples
#include <systemc.h>

class write_if : virtual public sc_interface{
public:
    virtual void write(int) = 0;
    virtual void reset() = 0;
};
```

```
class read_if : virtual public sc_interface{
public:
    virtual void read(int &) = 0;
    virtual int num_available() = 0;
};

class fifo : public sc_channel, public write_if, public read_if{
public:
    fifo(sc_module_name name) : sc_channel(name), num_elements(0),
first(0) {}
    void write(int c) {
        if (num_elements == max)
            wait(read_event);
        data[(first + num_elements) % max] = c;
        ++ num_elements;
        write_event.notify();
    }

    void read(int &c){
        if (num_elements == 0)
            wait(write_event);
        c = data[first];
        -- num_elements;
        first = (first + 1) % max;
        read_event.notify();
    }

    void reset() { num_elements = first = 0; }
    int num_available() { return num_elements;}

private:
    enum e { max = 10 };
    int data[max];
    int num_elements, first;
    sc_event write_event, read_event;
};

class fibonacci_gen : public sc_module{
public:
```

```
    sc_port<write_if> out;
    SC_HAS_PROCESS(fibonacci_gen);
    fibonacci_gen(sc_module_name name) : sc_module(name){
        SC_THREAD(main);
    }

    void main(){
        int first = 0, second = 1, next = 0;
        for (int i = 0; i < 10; i++){
            if(i <= 1)
                next = i;
            else {
                next = first + second;
                first = second;
                second = next;
            }
            out->write(next);
        }
    }
};

class consumer : public sc_module{
public:
    sc_port<read_if> in;
    SC_HAS_PROCESS(consumer);
    consumer(sc_module_name name) : sc_module(name) {
        SC_THREAD(main);
    }
    void main(){
        int c;
        cout << endl << "Fibonacci sequence: " << endl;
        while (true) {
            in->read(c);
            cout << c << endl;
            if(in->num_available() == 0)
                break;
        }
    }
};
```

```
class top : public sc_module{
public:
    fifo *fifo_inst;
    fibonacci_gen *prod_inst;
    consumer *cons_inst;
    top(sc_module_name name) : sc_module(name){
        fifo_inst = new fifo("Fifo1");
        prod_inst = new fibonacci_gen("FibonacciGen1");
        prod_inst->out(*fifo_inst);
        cons_inst = new consumer("Consumer1");
        cons_inst->in(*fifo_inst);
    }
};

int sc_main (int, char *[]) {
    top top1("Top1");
    sc_start();
    return 0;
}
```



```
Fibonacci sequence:
0
1
1
2
3
5
8
13
21
34
Finding VCD file...
```