ECE 5723 Final

By Bruce Huynh

12/08/2021

Problem 1:

In this problem we are to design a BFM for a circuit that calculates the Taylor series of cos(x).

Since this is a BFM, we are expected to have a cycle accurate model of our circuit. Creating a BFM also allows us to create a less detailed model. My circuit was based on the exponential BFM that the professor posted in Chapter 4.

```
SC_MODULE(cosTaylorSeries) {
    sc_in<sc_logic> clk, rst, start;
    sc_in<float> x;
    sc_out<float> out;
    sc_out<sc_logic> ready;
    enum state_types { INITIALIZE, WAIT_ON_START, ACTIVE, DONE};
    sc_signal<state_types> state;
    sc_signal<float> x_reg;


    double inBusval;

    SC_CTOR(cosTaylorSeries) {
        SC_THREAD(operation);
        sensitive << clk << rst;

    }

    void operation();


};
```

Here is the wrapping for my circuit. Similar to the professor's exponential BFM model, I turned the input x and output out into floats to simplify the internal parts. In the .cpp file, I made the circuit begin when a start pulse has finished. Afterwards it will do the calculation.

```
#include "5723problem1.h"



void cosTaylorSeries::operation() {

    int n = 12;
    int numClk = n + 2;
    float term = 1;
    float exp = 1;
    while (true) {

        if (rst == '1') {
            out = 0;
            state = INITIALIZE;

        }
        else if (clk == '1' && clk->event()) {
            switch (state) {
            case (INITIALIZE):

                if (start == '0') {
                    state = INITIALIZE;
                }
                else if (start == '1') {
                    state = WAIT_ON_START;
                }
                break;
            case (WAIT_ON_START):
                if (start == '1') {
                    state = WAIT_ON_START;
                }
                else if (start == '0') {
                    state = ACTIVE;
                }
                break;

```

```
38              case (ACTIVE):
39                  x_reg = x->read();
40                  wait(clk->posedge_event());
41
42                  for (int i = 1; i <= n; i++) {
43                      term = term * (-1) * x_reg.read() * x_reg.read() / (2 * i * (2 * i -1));
44                      exp = exp + term;
45
46                  }
47
48                  for (int i = 0; i < numClk - 1; i++)
49                      wait(clk->posedge_event());
50
51                  cout << "value of cos(" << x << ") is: " << exp << endl;
52                  out = exp;
53                  state = DONE;
54                  break;
55              case (DONE):
56                  ready = SC_LOGIC_1;
57                  state = INITIALIZE;
58
59                  cout << "number of iterations is " << n << endl;
60                  break;
61              }
62              }
63          wait();
64          }
65
66
67
68      }
69
```

```
case (ACTIVE):
    x_reg = x->read();
    wait(clk->posedge_event());

    for (int i = 1; i <= n; i++) {
        term = term * (-1) * x_reg.read() * x_reg.read() / (2 * i * (2 * i -1));
        exp = exp + term;

    }

    for (int i = 0; i < numClk - 1; i++)
        wait(clk->posedge_event());

    cout << "value of cos(" << x << ") is: " << exp << endl;
    out = exp;
    state = DONE;
    break;
```

This active case is where the calculation is being done. Besides doing just the calculations, we simulate the clock cycles as well. *n* is the number of iterations in our circuit and numClk is always $n + 2$ as we add a clk in the beginning and the end. The second for loop simulates the time it would take to calculate this.

Afterwards, we write our testbench:

```
4   SC_MODULE(Testbench) {
5       sc_signal<sc_logic> clk, rst, start;
6       sc_signal<float> x;
7       sc_signal<float> out;
8       sc_signal<sc_logic> ready;
9       cosTaylorSeries* CTS;
10
11
12
13
14      SC_CTOR(Testbench) {
15
16          CTS = new cosTaylorSeries("cosTaylorSeries");
17          (*CTS)(clk, rst, start, x, out, ready);
18
19
20          SC_THREAD(inputing);
21          SC_THREAD(reseting);
22          SC_THREAD(clocking);
23          sensitive << clk;
24
25          SC_THREAD(display)
26          sensitive << clk;
27      }
28
29
30      void inputing();
31      void reseting();
32      void clocking();
33      void display();
34
35
36  };
```

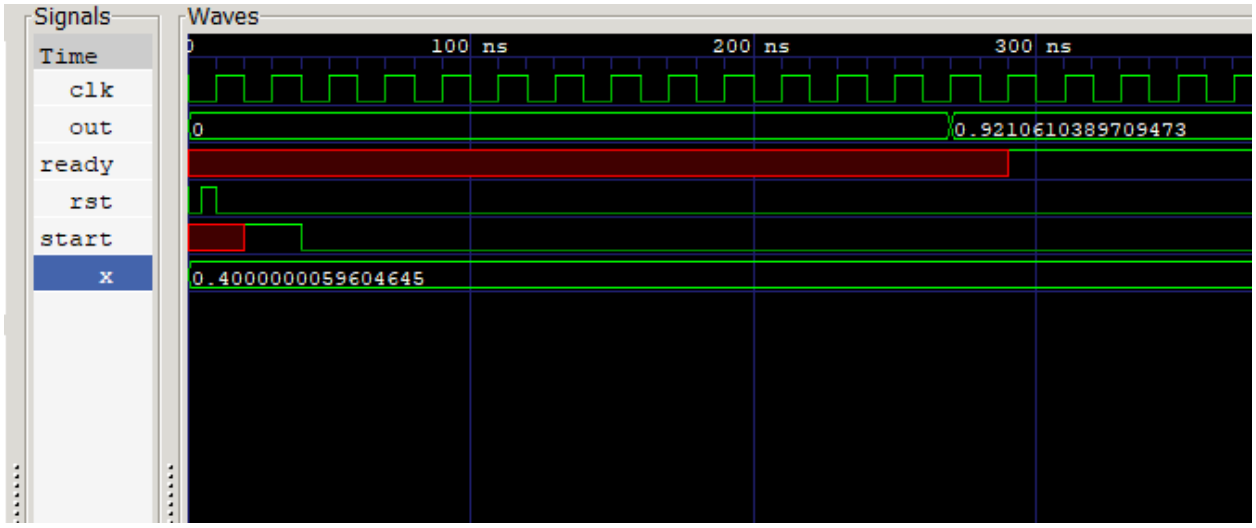
```
#include "Testbench.h"
#include <sysc/datatypes/bit/sc_lv.h>

void Testbench::inputing() {

    x = .4;
    wait(20, SC_NS);
    start = SC_LOGIC_1;
    wait(20, SC_NS);
    start = SC_LOGIC_0;
}


void Testbench::clocking() {
    while (true)
    {
        clk = sc_logic('0');
        wait(10, SC_NS);
        clk = sc_logic('1');
        wait(10, SC_NS);
    }
}

void Testbench::reseting() {
    rst = (sc_logic)'0';
    wait(5, SC_NS);
    rst = (sc_logic)'1';
    wait(5, SC_NS);
    rst = (sc_logic)'0';
};

void Testbench::display() {
    cout << "start = " << start << endl;
    cout << "x = " << x << endl;

    if (ready == SC_LOGIC_1) {
        cout << "cos(" << x << ") is " << out;
    }

};
```

Here are the results:

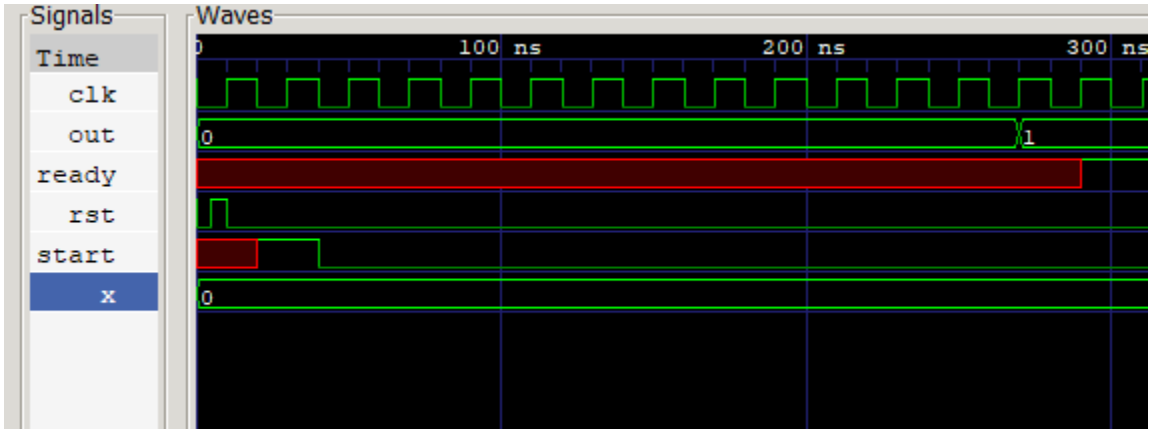
When $x = 0.4$ or 0.0110011 with 8 iterations, we get $\cos(x) = 0.921061$

```
value of cos(0.4) is: 0.921061
number of iterations is 8
```

This is very close to the expected value of 0.921060994. Here is what the waveform looks like with 8 iterations:



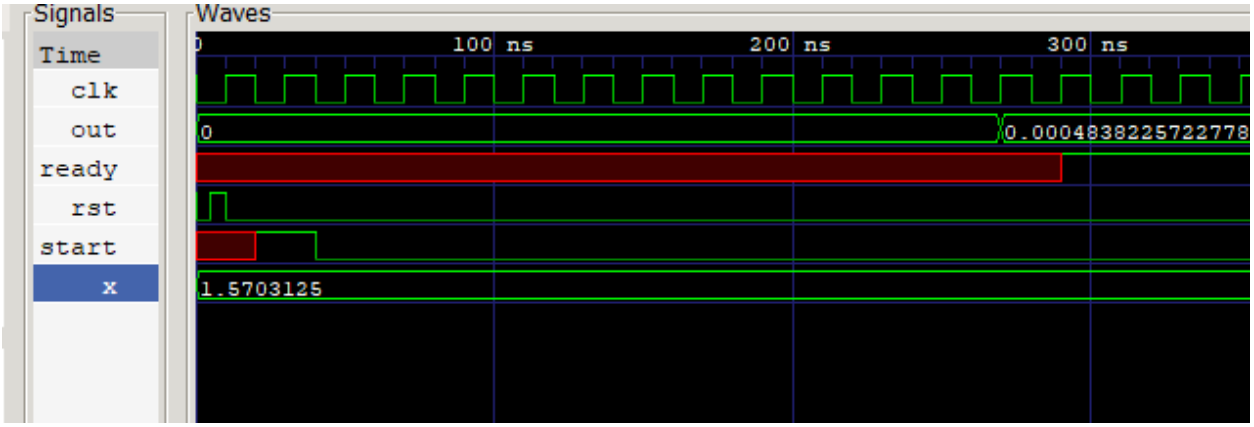
When x is equal to the smallest input value of 0, we get $\cos(x) = 1$, which is exactly the same as expected.



When x is equal to the largest input value of pi/2 or approximately 1.5703125 (1.1001001 is the closest we can get it to an 8-bit number), we get $\cos(x) = 0.000483$. This value is very close to the expected result of zero but not quite. We could test this with a larger number of iterations but 0.000483 is essentially 0 when you have only 8-bit resolution.

```
Info: (I702) default timescale unit used for tracing: 1 ps (cosTaylorSeries.vcd)
value of cos(1.57031) is: 0.000483823
number of iterations is 8
```

Here is the waveform for x = pi/2:



To choose the number of iterations I simulated these circuits with different *n* values knowing that I only have 8 bits to work with: 1 integer bit and 7 fractional bits. 8 iterations are too much as shown from *x* = pi/2 as the value is way too small to be recognized with 8-bits. So far around n = 3, 4 works the best for pi/2. They give similar values with 3 being negative and 4 being positive. When testing 4 iterations with the *x* = 0.4, we get a result of 0.921061 which is still very close to the actual value. *n* = 3, 4, 6, 8 gives the same value.

Now I will test a new number: 0.1055 (0.00011011)
The expected result is 0.994443.
Here are my results:
n = 3: 0.994443
n = 4: 0.994443
n = 6: 0.994443
n = 8: 0.994443

From these results, we can deduce that 3 is the number of iterations we need without the circuit saturating.

Problem 2:

In this problem, we are tasked with taking our non-blocking stack channel and rewriting it as a blocking stack channel.

In order to do this, we need to add *wait(<event>)* to both pushing and popping functions. I also turned the output of the function from *bool* to *void* as we are not going to be checking if there is data loss.

Here are the rewritten push and pop functions:

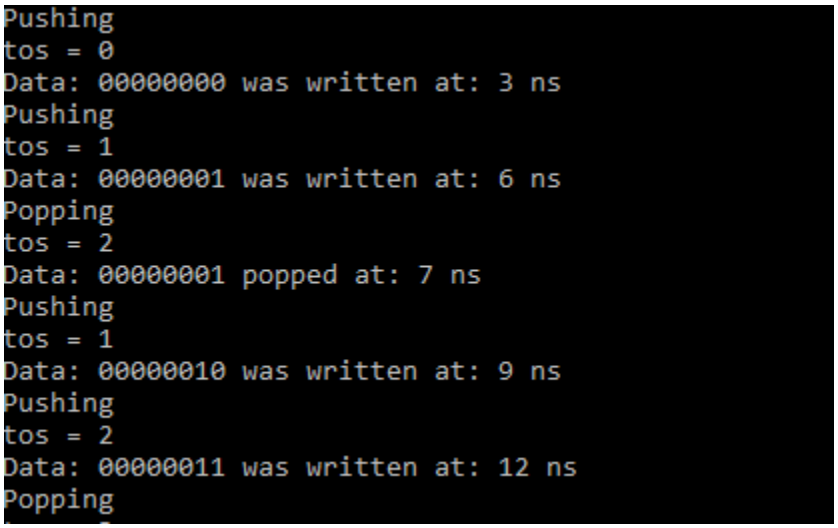
```
void stack::b_push(sc_lv<8> data) {
    cout << "Pushing" << endl;
    cout << "tos = " << tos << endl;
    if (tos == 17)
    {
        cout << "waiting for pop event" << endl;
        wait(pop_event);
        contents[tos++] = data;

        push_event.notify();
        }
    else if (tos < 17){
        contents[tos++] = data;
    }

}
```

In our push function, if it is full then it will wait for a pop event to free up a space on top of the stack. Then it will add the data in the stack. Otherwise if the stack has free spots, it will put the data in the next available spot.

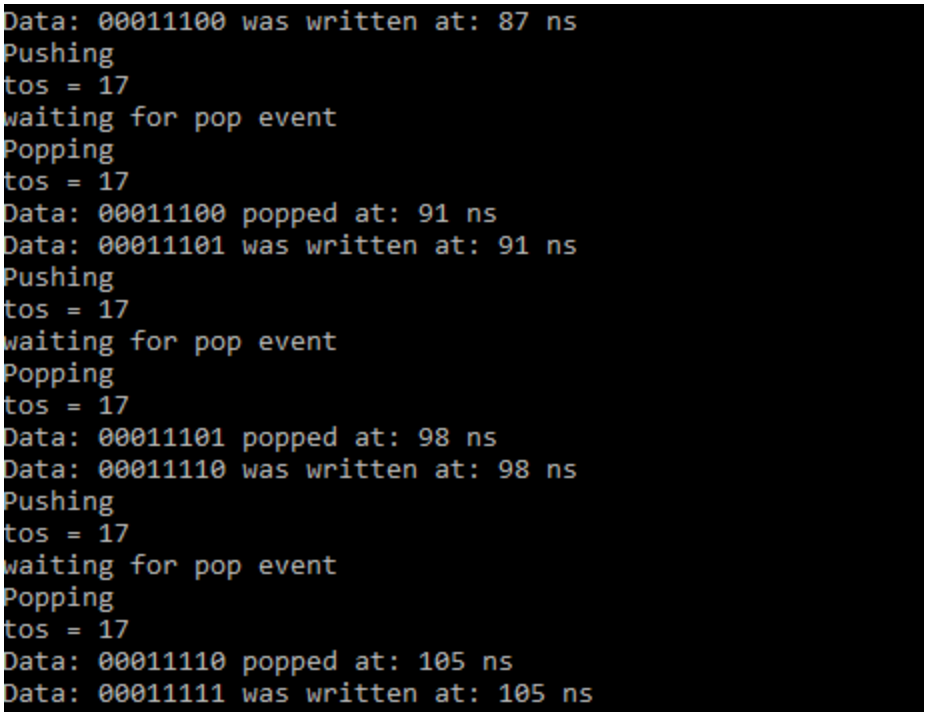
```
void stack::b_pop(sc_lv<8>& data) {
    cout << "Popping" << endl;
    cout << "tos = " << tos << endl;
    if (tos == 0)
    {
        cout << "waiting for push event" << endl;
        wait(push_event);
        data = contents[--tos];
        pop_event.notify();
        }
    else if (tos > 0) {
        data = contents[--tos];
        pop_event.notify();
    }

}
```

In our pop function, if there is nothing to pop then it will wait for a push event to give it something to pop. Otherwise, it will pop an item on the stack and let the push event know.

Testbench:

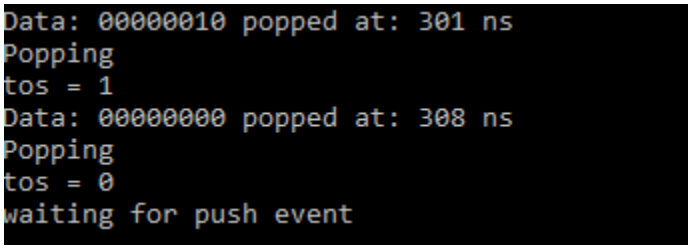


First, we push all the data onto the stack on top of each other. “tos” represents the index on the stack where the data is being put. However, our stack can only hold 18 values, so when it gets to index 17, it has to pop data.



In this image, you can see that the data cannot be written in when the stack is full. We wait for the data to be popped and then we put the new data in. This image also confirms that our stack works as a stack as the previous value that was written in gets popped next time data wants to be written in showing that it is a LIFO.

Lastly, when there is nothing left to pop, it will wait for more data to be pushed before it does anything.



Problem 3:

In this problem, we are tasked with creating a TLM system that has two initiators connected to a single target through an interconnect. The following is the .h file of my interconnect:

```
#include <systemc.h>
#include "tlm_utils/simple_initiator_socket.h"
#include "tlm_utils/simple_target_socket.h"
#include "tlm.h"
#include <iostream>
using std::cout;


class interconnect : public sc_module {
public:
    sc_mutex bus;
    tlm_utils::simple_target_socket_tagged<interconnect, 32> target_socket[2];
    tlm_utils::simple_initiator_socket<interconnect, 32> initiator_socket;



public:
    //SC_CTOR(interconnect) : target_socket("target_socket"){
        interconnect(sc_core::sc_module_name name) :
           sc_core::sc_module(name), nBlockWriteRead(0) {

               target_socket[0].register_nb_transport_fw(this, &interconnect::nb_transport_fw, 0);
               target_socket[1].register_nb_transport_fw(this, &interconnect::nb_transport_fw, 1);

               //for (int i = 0; i < 256; i++)
               //    memArray[i] = (sc_lv<8>) (i % 256 + 192);
               //nBlockWriteRead = new tlm::tlm_generic_payload();
               //for (int i = 0; i < 4; i++) *(data + i) = i + 192;
               //SC_THREAD(nbMemWR);
        }

    //void thread_1();
    //void thread_2();
    sc_lv<8> data[5];
    sc_lv<8> memArray[256];
    tlm::tlm_generic_payload* nBlockWriteRead;
    void doSomethingGood(tlm::tlm_generic_payload&, sc_time);
    void nbMemWR();
    virtual tlm::tlm_sync_enum nb_transport_fw(int SocketID, tlm::tlm_generic_payload&, tlm::tlm_phase&, sc_time&);
};
```

Our interconnect would need two target sockets (coming from the initiators) and one initiator socket (going to target):

```
tlm_utils::simple_target_socket_tagged<interconnect, 32> target_socket[2];
tlm_utils::simple_initiator_socket<interconnect, 32> initiator_socket;
```

I made them the size of the data, which is 32 bits and I made two instances of the target socket.

Then, we need to connect our target sockets with the nb_transport_fw call:

```
interconnect(sc_core::sc_module_name name) :
   sc_core::sc_module(name), nBlockWriteRead(0) {

      target_socket[0].register_nb_transport_fw(this, &interconnect::nb_transport_fw, 0);
      target_socket[1].register_nb_transport_fw(this, &interconnect::nb_transport_fw, 1);
```

This call is exactly the same as the one found in the memoryUnit file with two big differences: I added an argument for the ID of the target socket to be able to choose and tell what socket is receiving information. The actual locking of the data transmission itself happens in the nbMemWR() function.

After all that is done, we can create our testbench with two initiators, one target and one interconnect.

```
#include "memWriterReader.h"
#include "memoryUnit.h"
#include "interconnect.h"

SC_MODULE(memWriterReader_TB)
{
    memWriterReader *WR1;
    memWriterReader* WR2;
    memoryUnit *MU1;
    interconnect *IN1;

    SC_CTOR(memWriterReader_TB)
    {
        WR1 = new memWriterReader("WR1");
        WR2 = new memWriterReader("WR2");
        MU1 = new memoryUnit("memory");
        IN1 = new interconnect("interconnect");
        //WR1->memWRSocket.bind(MU1->memSocket);
        WR1->memWRSocket.bind(IN1->target_socket[0]);
        WR2->memWRSocket.bind(IN1->target_socket[1]);
        IN1->initiator_socket.bind(MU1->memSocket);

    }
};
```

From this screenshot, we can see how these sockets are wired. The output of the initiators are connected to the target sockets of the interconnect. Then the interconnect has an initiator socket connecting to the memSocket which finishes our module.

Here is how it looks after running and printing:

```
W, @0 data:00000101 00000110 00000111 00001000 00001001  @time 0 s delay=0 s
Current target = target[0]
Above was completed @time 123 ns
W, @11 data:00010000 00010001 00010010 00010011 00010100  @time 0 s delay=123 ns
Current target = target[0]
Above was completed @time 246 ns
R, @22 data:00010000 00010001 00010010 00010011 00010100  @time 0 s delay=246 ns
Current target = target[0]
Above was completed @time 369 ns
R, @33 data:00010000 00010001 00010010 00010011 00010100  @time 0 s delay=369 ns
Current target = target[0]
Above was completed @time 492 ns
W, @44 data:00110001 00110010 00110011 00110100 00110101  @time 0 s delay=492 ns
Current target = target[0]
Above was completed @time 615 ns
R, @55 data:00110001 00110010 00110011 00110100 00110101  @time 0 s delay=615 ns
Current target = target[0]
Above was completed @time 738 ns
R, @66 data:00110001 00110010 00110011 00110100 00110101  @time 0 s delay=738 ns
Current target = target[0]
Above was completed @time 861 ns
R, @77 data:00110001 00110010 00110011 00110100 00110101  @time 0 s delay=861 ns
Current target = target[0]
Above was completed @time 984 ns
R, @88 data:00110001 00110010 00110011 00110100 00110101  @time 0 s delay=984 ns
Current target = target[0]
Above was completed @time 1107 ns
R, @99 data:00110001 00110010 00110011 00110100 00110101  @time 0 s delay=1107 ns
Current target = target[0]
Above was completed @time 1230 ns
W, @110 data:01110011 01110100 01110101 01110110 01110111  @time 0 s delay=1230 ns
Current target = target[0]
```

This shows the first target socket being used only.

```
W, @110 data:01110011 01110100 01110101 01110110 01110111  @time 0 s delay=1230 ns
Current target = target[0]
Above was completed @time 1353 ns
W, @0 data:00000101 00000110 00000111 00001000 00001001  @time 0 s delay=0 s
Current target = target[1]
Above was completed @time 123 ns
W, @11 data:00010000 00010001 00010010 00010011 00010100  @time 0 s delay=123 ns
Current target = target[1]
Above was completed @time 246 ns
R, @22 data:00010000 00010001 00010010 00010011 00010100  @time 0 s delay=246 ns
Current target = target[1]
Above was completed @time 369 ns
R, @33 data:00010000 00010001 00010010 00010011 00010100  @time 0 s delay=369 ns
Current target = target[1]
Above was completed @time 492 ns
W, @44 data:00110001 00110010 00110011 00110100 00110101  @time 0 s delay=492 ns
Current target = target[1]
Above was completed @time 615 ns
R, @55 data:00110001 00110010 00110011 00110100 00110101  @time 0 s delay=615 ns
Current target = target[1]
Above was completed @time 738 ns
R, @66 data:00110001 00110010 00110011 00110100 00110101  @time 0 s delay=738 ns
Current target = target[1]
Above was completed @time 861 ns
R, @77 data:00110001 00110010 00110011 00110100 00110101  @time 0 s delay=861 ns
Current target = target[1]
Above was completed @time 984 ns
R, @88 data:00110001 00110010 00110011 00110100 00110101  @time 0 s delay=984 ns
Current target = target[1]
Above was completed @time 1107 ns
R, @99 data:00110001 00110010 00110011 00110100 00110101  @time 0 s delay=1107 ns
Current target = target[1]
Above was completed @time 1230 ns
W, @110 data:01110011 01110100 01110101 01110110 01110111  @time 0 s delay=1230 ns
Current target = target[1]
Above was completed @time 1353 ns
```

Afterwards, the second target is free to communicate its data.