NTNU
Norwegian University of
Science and Technology

Exercise 3 System verification in SystemC

Ingebrigt Bartholsen, Erlend Strandvik

Exercise paper in course TFE4170 System on Chip

Norwegian University of Science and Technology

February 2014

NORWEGIAN UNIVERSITY OF SCIENCE AND TECHNOLOGY

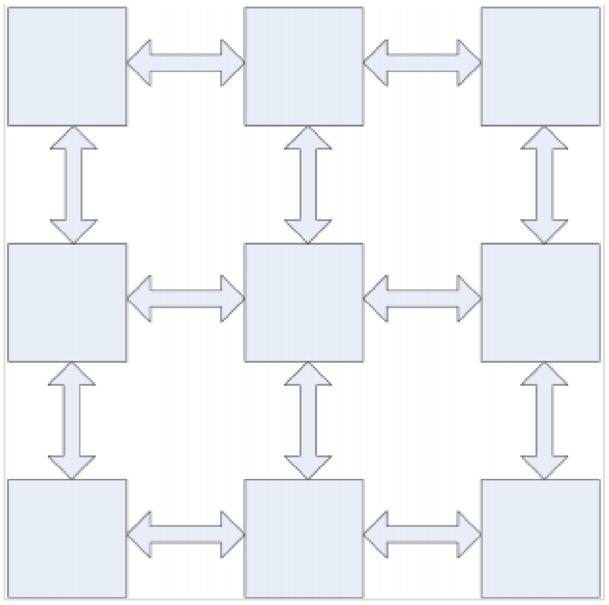
Group 1

Exercise report
Exercise 3 System verification in SystemC

Contents

1 Introduction

In this exercise a communication protocol between a square number of n-processors are to be implemented. The idea behind this is that each processor only communicates between himself and his immediate neighbours. So that the system in itself does not need any shared buses, but lets messages and information flow by pushing them trough a path of processors. Below you can see a processor array consisting of $n = 9$ processors.

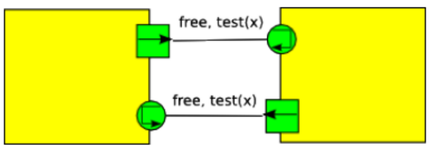


First the interface got implemented and the system above tested. Then the system was analyzed and improved. Finally the scalability of the system was evaluated.

2 Tasks

2.1 Task 1 -Interface and processor

Below you can see the interface of the processor core. It consists of a $free()$- and a $test(intnumber)$-function. The $free()$-function is implemented to check if a processor is available for testing. $free()$ is implemented with mutexes in order to prevent multiple processors setting the $isFree$-flag simultaneously. The $test(intnumber)$-functions checks if the processor running the test has the same number as its neighbours. The test function is branched out through the system of processors as a result of the processors being tested upon, tests on their neighbours as well.



```
class processor_if:  virtual public sc_interface {
public:
virtual bool free() = 0;
virtual bool test(int) = 0;
virtual bool leftTest(int) = 0;
virtual bool upTest(int) = 0;
virtual void MCU_test() = 0;
virtual void generateRandomNumber() = 0;
bool isFree;
private:
};

//if there's a connection to the up and the function to the up is available.
//then test
bool Processor::upTest(int number){
if (upConn){
while(!(up->isFree)){
wait(100*MS);
cout << "Waiting for up to be free" << endl;
}
return up->test(number);
} else
return false;
}

//if there's a connection to the left and the function to the left is available.
//then test
bool Processor::leftTest(int number){
if (leftConn){
while(!(left->isFree)){
cout << "Waiting for left to be free" << endl;
wait(100*MS);
}
return left->test(number);
} else
return false;
}

//returns true if the number exists either here, up or to the left.
bool Processor::test(int number){
isFree = false; //Occupied by another MCU.
//Testing our own number, and up and left if there is a connection there.
if(id == number || (upConn && upTest(number)) || (leftConn && leftTest(number))){
isFree = true;
cout << "Returning true on value: " << number << " my id: " << id << endl;
return true;
} else {
isFree = true;
cout << "Returning false on value: " << number << ". My id: " << id << endl;
return  false;


//checking if someone else is using the MCU
bool Processor::free(){
while(process_access.trylock() != 0){

cout << "Processor " << id << " is waiting on mutex." << endl;
wait(mutexdelay);
}
process_access.lock(); //Using mutex to prevent multiple processors setting the isFr
if(isFree){
isFree = false; //setting a busy flag for other processors to see
process_access.unlock();
return true; //Telling processor its available for testing
}
else{
process_access.unlock();
return false; //not available for testing
}
}
```

2.2 Task 2 -System test

In task 2 a 3x3 matrix of processors were generated, each generating a random number upon construction. Then testing with its neighbours if it chose a available number. As mentioned over, the test branches out through the matrix in order to obtain unique numbers for all the processors. The test sequence given by the task states that the processors are to check its neighbours up and to the left. By generating random test sequences in our testbench we soon found out the algorithm for testing only the ones up and to the left does not work. An example of a processor which can obtain a non-unique number is the processor to the bottom left of the array, as well as the one to the right at the top. These are examples of processor that will never be in contact with many other processors if we only test upwards and to the left.

```
class top : public sc_module {
public:
virtual void testbed();
Processor* procarray[9];
top (sc_module_name name, int numberOfProcessors) : sc_module (name){
char buf[20];
for (int i = 0; i < numberOfProcessors; ++i){
sprintf(buf, "MCUinst%d\0", i);
procarray[i] = new Processor(buf);
}
int row = 0;
int nSqrt = sqrt(numberOfProcessors);
int lastRow = nSqrt-1;
cout << "Connecting processor array: " << endl;//Underneath the code for connecting
for (int i = 0; i < numberOfProcessors; i++){
// Connection upwards:
if (row != 0){
procarray[i]->up(*procarray[i-nSqrt]);
cout << i << " connected upwards to " << i-nSqrt << endl;
}
// Connection downwards:
if (row != lastRow){
procarray[i]->down(*procarray[i+nSqrt]);
cout << i << " connected downwards to " << i+nSqrt << endl;
}
// Connection left:
if (i % nSqrt != 0){
procarray[i]->left(*procarray[i-1]);
cout << i << " connected left to " << i-1 << endl;
}
//Connection right:
if (i % nSqrt != nSqrt - 1){
procarray[i]->right(*procarray[i+1]);
cout << i << " connected right to " << i+1 << endl;
} else
row++;
}
SC_HAS_PROCESS(top);
SC_THREAD(testbed);
}

};

void top::testbed(){
wait(1000*ms);
for (int i = 0; i < numberOfProcessors; i++){
//int tmp = rand()%numberOfProcessors;
int tmp = numberOfProcessors-i-1;
procarray[tmp]->test_signal = true;
cout << "Testing for procarray[" << tmp << "]" << endl;
wait(500*ms);
}
for (int i = 0; i < numberOfProcessors; ++i){
cout << "Processor number " << i << " has id: " << procarray[i]->id << endl;
}
}
```

2.3 Task 3 -Analysis

In task 3 we wanted to analyse the algorithm used in task 2 in order to make a new and improved algorithm. The first thing we noticed was that some of the processors was partially isolated from others when only up and left testing was available. Secondly it was a slow method. When scaled up, it would create a lot of extra test branches. Therefore we wanted to create an algorithm that reached all the corners of the processor array, and also had a better running time. Our new algorithm starts with the first position in the array and generates a random number. When a number has been generated, it puts a flag $isFree = true$. The next processor in the array listens in on this flag, and as soon as it evaluates as true, it generates a random number, and checks with the first processor if this value is taken. The flag procedure is then repeated and alerts the next processor in the waiting chain. What is important to notice is that this is a chain of adjacent processors. So in the first row, all processors listen through its left port, i.e. 2 listens to 1. Therefore the last processor in the first row is listened on through the up-port of the last processor in the next row, and the second row listens through the right ports where the last processor is the first in the second row chain. This means that the processor chain of testing, ”snakes” through the array of processor and calls the test function in each processor only once(which branches through the previous processors).

```
//returns true if the number exists either here, up or to the left.
bool Processor::test(int number){
if (id == number){
return true;
}else if (pos == 0){
return false;
}else if (!(row%2) && collumn != 0){
return leftTest(number);
} else if (!(row%2) && collumn == 0){
return upTest(number);
} else if((row%2) && (collumn == (sqrt(total)-1))){
return upTest(number);
}else if(row%2 && collumn != (sqrt(total)-1)){
return rightTest(number);
}
}
```

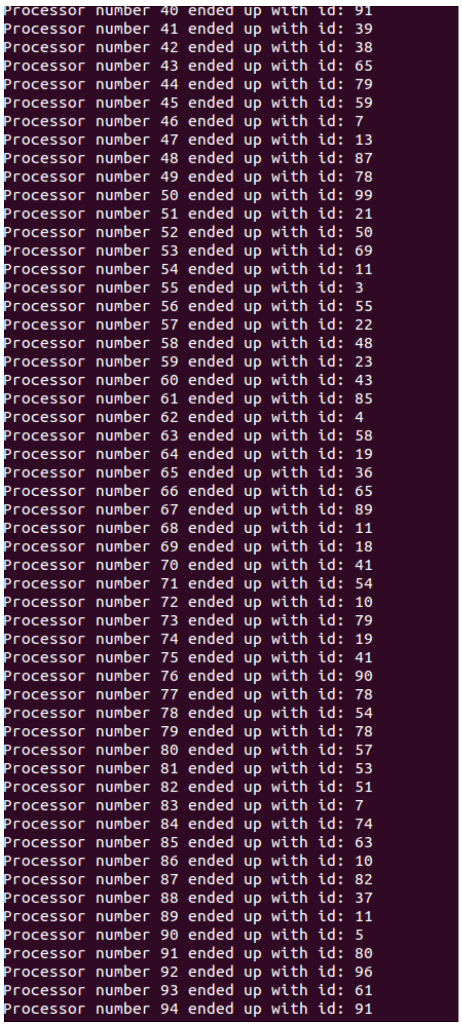
The new MCU_test-function can be seen below.

```
void Processor::MCU_test(){
generateRandomNumber();
row = pos / sqrt(total);
collumn = pos% (int)sqrt(total);
upConn = up.get_interface(); // Checks the connection upwards, 0 = unconnected
leftConn = left.get_interface();// Checks the connection left, 0 = unconnected
rightConn = right.get_interface();// Checks the connection left, 0 = unconnected
bool exists = false;
wait(test_signal.value_changed_event());
if (pos == 0){
cout << "pos 0 is free!!!!!" << endl;
isFree = true;
cout << isFree;
}else if (!(row%2) && collumn != 0){
cout << "Processor " << pos << " running left tests with id " << id << endl;
exists = leftTest(id);
while(exists){
generateRandomNumber();
cout << "Processor " << pos << " still running left tests with id " << id << endl;
exists = leftTest(id);

}
cout << "pos " << pos << " is free!!!!!" << endl;
} else if (!(row%2) && collumn == 0){
exists = upTest(id);
while(exists){
generateRandomNumber();
exists = upTest(id);
cout << "Processor " << pos << " still running up tests with id " << id << endl;
}
cout << "pos " << pos << " is free!!!!!" << endl;
} else if(row%2 && (collumn == (sqrt(total)-1))){
exists = upTest(id);
while(exists){
generateRandomNumber();
exists = upTest(id);
cout << "Processor " << pos << " still running up tests with id " << id << endl;
}
cout << "pos " << pos << " is free!!!!!" << endl;
}else if(row%2 && collumn != (sqrt(total)-1)){
exists = rightTest(id);
while(exists){
generateRandomNumber();
exists = rightTest(id);
cout << "Processor " << pos << " still running right tests with id " << id << endl;
}
cout << "pos " << pos << " is free!!!!!" << endl;
}
isFree = true;
}
```

2.4 Task 4 -Scalability

Since the implemented working algorithm in task3 only called maximum one function to find out if the number existed, this function would scale better than the up and left test function from task 2. Entering $numberOfProcessors = 100$ showed that it also with a hundred processors did compute correct answers. Below you can see a screenshot of most of the $numberOfProcessors = 100$ processors. (could not fit all).



```
Processor number 40 ended up with id: 91
Processor number 41 ended up with id: 39
Processor number 42 ended up with id: 38
Processor number 43 ended up with id: 65
Processor number 44 ended up with id: 79
Processor number 45 ended up with id: 59
Processor number 46 ended up with id: 7
Processor number 47 ended up with id: 13
Processor number 48 ended up with id: 87
Processor number 49 ended up with id: 78
Processor number 50 ended up with id: 99
Processor number 51 ended up with id: 21
Processor number 52 ended up with id: 50
Processor number 53 ended up with id: 69
Processor number 54 ended up with id: 11
Processor number 55 ended up with id: 3
Processor number 56 ended up with id: 55
Processor number 57 ended up with id: 22
Processor number 58 ended up with id: 48
Processor number 59 ended up with id: 23
Processor number 60 ended up with id: 43
Processor number 61 ended up with id: 85
Processor number 62 ended up with id: 4
Processor number 63 ended up with id: 58
Processor number 64 ended up with id: 19
Processor number 65 ended up with id: 36
Processor number 66 ended up with id: 65
Processor number 67 ended up with id: 89
Processor number 68 ended up with id: 11
Processor number 69 ended up with id: 18
Processor number 70 ended up with id: 41
Processor number 71 ended up with id: 54
Processor number 72 ended up with id: 10
Processor number 73 ended up with id: 79
Processor number 74 ended up with id: 19
Processor number 75 ended up with id: 41
Processor number 76 ended up with id: 90
Processor number 77 ended up with id: 78
Processor number 78 ended up with id: 54
Processor number 79 ended up with id: 78
Processor number 80 ended up with id: 57
Processor number 81 ended up with id: 53
Processor number 82 ended up with id: 51
Processor number 83 ended up with id: 7
Processor number 84 ended up with id: 74
Processor number 85 ended up with id: 63
Processor number 86 ended up with id: 10
Processor number 87 ended up with id: 82
Processor number 88 ended up with id: 37
Processor number 89 ended up with id: 11
Processor number 90 ended up with id: 5
Processor number 91 ended up with id: 80
Processor number 92 ended up with id: 96
Processor number 93 ended up with id: 61
Processor number 94 ended up with id: 91
```