|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Document no. |  | | | | | | | TFE4171 – Design of Digital Systems 2 |
| TFE4171-Ex5-01 |
| Revision | | Date | | | | |
| 1.0 | | 14.04.2015 | | | | |
| Revision By | |  | | | | |
| Hope / Grindheim | |  | | | | |
| ­ | | | | | | | |
| Laboratory Report | | | | | |  | |
| Exercise V | | | | | |  |  |
|  | | | | | | |
| ­ | | | | | | | | |
| Author(s):  B.Hope  S.Grindheim | |  | |  |  | | | |
| Revision Notes | * 1.0: Initial release | | | |
|  |  | | | |
|  |  | | | |
|  |  | | | |
|  |  | | | |
| ***Bengt Erik Hope / Svein Grindheim © 2015 Bergen, NORWAY*** | | | | | | | | |

**🗎Page intentionally left blank**

* Abstract

*This document attempts to answer the questions as listed in the fifth and last exercise in the NTNU course TFE4171 – Design of Digital Systems 2. The objectives of the exercise is to learn the basics of SystemC.*

Contents

[**A short note about toolset and deliverables** iii](#_Toc416798436)

[Assignment 1 – Naïve FIFO 1](#_Toc416798437)

[Assignment 2 – Blocking FIFO 2](#_Toc416798438)

[Assignment 3 – Controlled Interface 3](#_Toc416798439)

[Assignment 4 – Handshake Control 5](#_Toc416798440)

#### A short note about toolset and deliverables

We choose to work on Windows and with SystemC version 2.3.1. An archive containing all source code is attached to this document. It also contains binaries built with MS VC++11 for each task. We ask that the reader refer to the source code delivered if there is any questions not specifically mentioned in this paper.

# – Naïve FIFO

A trial and error approach was used in order to find the smallest FIFO size for channel delays of and . We can see from our results – as depicted in TABLE 1, that the best FIFO sizes would be 39 and 20 respectively, which in turn determines the approx. percentage of unsafe operations to 24% and 10%. The percentage can be confirmed using the following formulae:

**OR**

Based on these figures – and with such a large proportion of unsafe operations, one can conclude that this model is not particularly favorable. Although this is to be considered a performance model which need not necessarily render realistic behavior, one would perhaps desire a more stable behavior.

Again referring to TABLE 1: to guarantee error free transmissions, the buffer size must be increased to at least 377 – which is a fairly large size compared to previous discussions. We do get the average transfer time down to almost 100ns (100.932ns) however.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| FIFO Size | Avg. Transfer Time | Overwrites | Empty Reads | Transfers | Unsafe Operations |
| 20 | 124.22ns | 9378 | 10298 | 81235 | 24,2 % |
| 39 | 109.685ns | 3989 | 4922 | 92000 | 9,7 % |
| 377 | 100.932ns | 0 | 933 | 99978 | 0,9 % |

Table 1

The channel model is based on non-blocking and non-controlled read/write interface. We see that the best case will always have 933 empty reads no matter the scenario which might be considered a fairly weak result. Even though a non-blocking is favorable to a blocking scenario for many reasons (for example wait-freedom is a strong non-blocking guarantee of progress, which also prevents starvation), improving the non-blocking algorithm should at the very least be considered. Perhaps one should however consider introducing a blocking scenario where checking whether the FIFO is full/empty will lead to a more realistic model to prevent empty reads and in general reduce the required buffer size in order to disallow unsafe operations.

# – Blocking FIFO

To introduce a blocking interface fifo\_write\_if**<**char**>** and fifo\_read\_if**<**char**>** was replaced with fifo\_write\_bl\_if**<**char**>** and fifo\_read\_bl\_if**<**char**>** respectively where needed in the files *producer.h*, *consumer.h* and *fifo.h*. In *fifo.h* sc\_event event\_write, event\_read; was added so we have access to read/write events in the .cpp implementation file (please refer to the attached archive containing source files for further reference regarding the above changes).

The write/read functions in *fifo.cpp* was implemented as follows (changes/additions highlighted):

[…]

/\* WRITE-METHOD IMPLEMENTATION \*/

void fifo::write (char c)

{

// Write byte into the fifo buffer

data[(first + num\_elements) % size] = c;

**//fire write event**

**event\_write.notify();**

#ifdef TRACE

cout << "+" << c;

#endif

// if the buffer is already full wait for read event

if (num\_elements == size) //++ num\_overwrite;

**wait(event\_read);**

else // otherwise we increment the num of elements counter

++ num\_elements;

}

/\* READ METHOD IMPLEMENTATION \*/

char fifo::read ()

{

**//prohibit empty reads**

**if (!num\_elements)**

**wait(event\_write);**

// local buffer for storing the read byte until return

char c;

// record time

last\_time = sc\_time\_stamp();

// fetch the "oldest" byte in the buffer

c = data[first];

// compute the new stats based on new counters

compute\_stats();

// if the buffer is already empty, we update the empty read counter

if (num\_elements == 0) {

c = '0';

++ num\_emptyread;

}

// otherwise we decrement the number of elements counter, and

// increment the first pointer to point to the next oldest element

else {

-- num\_elements;

first = (first + 1) % size;

}

#ifdef TRACE

cout << "-" << c;

#endif

**//fire read event**

**event\_read.notify();**

// return the correct byte

return c;

}

[…]

Code Excerpt 1

The result as requested in the assignment text – namely finding the buffer size for 125ns and 110ns – is depicted in the table below:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| FIFO Size | Avg. Transfer Time | Overwrites | Empty Reads | Transfers | Unsafe Operations |
| 11 | 124.053ns | 0 | 0 | 87 641 | 0 % |
| 22 | 109.502ns | 0 | 0 | 94 919 | 0 % |

Table 2

Our conclusion proves a point in the final discussions in the last assignment, namely that implementing blocking features to prohibit overwriting and “empty reads” greatly reduced the required buffer size compared to the naive FIFO. There are also no unsafe operations!

# – Controlled Interface

We now revert back to using non-blocking FIFO, but with controlled read/write interface. The source code from the first assignment is used as a backbone.

First fifo\_write\_if**<**char**>** and fifo\_read\_if**<**char**>** is replaced with fifo\_write\_ctrl\_if**<**char**>** and fifo\_read\_ctrl\_if**<**char**>** in files *producer.h,* *consumer.h* and *fifo.h* were necessary.

Two new functions, full() and empty(), was defined in *fifo.cpp* in order to flag on full/empty buffer:

[…]

bool fifo::empty()

{

if (num\_elements == 0)

return true;

else

return false;

}

bool fifo::full()

{

if (num\_elements == size)

return true;

else

return false;

}

[…]

Code Excerpt 2

We then made use of the buffer check in CODE EXCERPT 2 in the read/write source code for the calling masters.

*consumer.h* therefore has the following additions:

[…]

// thread needs to run indefinitely until simulator itself stops

while (true)

{

// read byte from the readp port which we assume is connected to

// the fifo's fifo\_read\_if interface

**if (!readp->empty())**

c = readp->read ();

[…]

Code Excerpt 3

While *producer.h* has the following additions:

[…]

// and then we send each byte to the fifo as fast as possible

while (--i >= 0)

{

**// sends one byte and increments the pointer in one turn**

**if(!writep->full())**

writep->write (\*p++);

[…]

Code Excerpt 4

The result as requested in the assignment text is depicted in the table below:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| FIFO Size | Avg. Transfer Time | Overwrites | Empty Reads | Transfers | Unsafe Operations |
| 12 | 124.622ns | 0 | 0 | 92 153 | 0 % |
| 23 | 109.503ns | 0 | 0 | 91 692 | 0 % |

Table 3

We notice that in our efforts to avoid using a blocking based read/write solution we have successfully managed to prevent any unsafe operations in a non-blocking solution. The results is however not very different from the ones in our previous assignment, we note that we do need a slightly bigger buffer to achieve the same transfer time in the current solution.

# – Handshake Control

We now get ready to use handshake control. The source code from the first assignment is again used as a backbone for the implementation in this assignment. We refer the reader to the attached source code for detailed reference as there is significant changes to the code and only a selection will be shown in this section.

First fifo\_write\_if**<**char**>** and fifo\_read\_if**<**char**>** is replaced with fifo\_write\_cb\_if**<**char**>** and fifo\_read\_cb\_if**<**char**>** in files *producer.h,* *consumer.h* and *fifo.h* were necessary. The consumer and producer classes was both made to inherit from the fifo\_callback\_if class. We then implemented the virtual void ack(bool) function as follows:

[…]

/\*PRODUCER ack\*/

void ack(bool c)

{

request\_write\_ = c;

}

[…]

/\*CONSUMER ack\*/

void ack(bool c)

{

request\_read\_ = c;

}

[…]

Code Excerpt 5

The actual handshake control happens in the main() of *producer.h* and *consumer.h* as follows:

[…]

/\*…from PRODUCER’s main()\*/

while (true)

{

int i = 1 + rand() % 19; // 1 <= i <= 19

// and then we send each byte to the fifo as fast as possible

while (--i >= 0)

{

**while (!request\_write\_)**

**{**

**writep->req\_w(true);**

**wait(short\_time);**

**}**

// sends one byte and increments the pointer in one turn

writep->write (\*p++);

**writep->req\_w(false);**

// if string end is reached, start at the beginning of the

// string again

if (!\*p) p = str;

// decrement the total count

-- total;

}

[…]

/\*…from CONSUMER’s main()\*/

while (true)

{

**while (!request\_read\_)**

**{**

**readp->req\_r(true);**

wait(short\_time);

**}**

c = readp->read ();

**readp->req\_r(false);**

wait (long\_time);

}

[…]

Code Excerpt 6

For the FIFO implementation we had to declare callback ports – specifically

sc\_port <fifo\_callback\_if> reqack\_readp and sc\_port <fifo\_callback\_if> reqack\_writep, in *fifo.h* which could be instantiated on top.cpp. We also added the req\_r/w(bool) as virtual functions, so we could define them as additional read/write logic in fifo.cpp:

[…]

/\*DEFINE REQUEST FUNCTIONS\*/

void fifo::req\_w (bool req\_write)

{

/\*request writing to buffer\*/

if(req\_write)

reqack\_writep->ack(num\_elements < size);

/\*remove request\*/

else

reqack\_writep->ack(false);

}

void fifo::req\_r (bool req\_read)

{

/\*request to read from buffer\*/

if(req\_read)

reqack\_readp->ack(num\_elements > 0);

/\*remove request\*/

else

reqack\_readp->ack(false);

}

[…]

Code Excerpt 7

The result as requested in the assignment text is depicted in the table below:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| FIFO Size | Avg. Transfer Time | Overwrites | Empty Reads | Transfers | Unsafe Operations |
| 10 | 122.861ns | 0 | 0 | 100 000 | 0 % |
| 22 | 109.675ns | 0 | 0 | 99 988 | 0 % |

Table 4

We now have the best channel model as of yet because the system is not blocked at any time.

A decrease in the needed buffer size was achieved and more transfers are allowed to take place.