Exercise 1: SystemC and Virtual Prototyping

Setting Up Your Environment

Lukas Steiner
WS 2024/2025

Prerequisites for Windows Users

Installing Windows Subsystem for Linux (WSL)

On Windows platforms we recommend using the **Windows Subsystem for Linux** (WSL). General information about the WSL can be found here. Information on the installation can be found here. To keep things unified please choose **Ubuntu** or **Debian** as your Linux distribution.

Running Graphical Applications on WSL

In order to run applications with a graphical user interface you have to set up **X11 forwarding** for the WSL. Download and install VcXsrv. Run XLaunch, check the box *Disable access control* in the *Extra settings* window and save your configuration. If you want VcXsrv to be automatically started at system startup create a shortcut of your saved configuration and add it to the startup folder (press Windows key + R and type `shell:startup` to open the folder).

WSL 1 Specific Setup

Add the following line to the `.bashrc` file (located in the Linux distribution's home directory) and restart the WSL.

```
export DISPLAY=:0
```

WSL 2 Specific Setup

Add the following two lines to the `.bashrc` file (located in the Linux distribution's home directory) and restart the WSL.

```
export DISPLAY=$(awk '/nameserver / {print $2; exit}' \
    /etc/resolv.conf 2>/dev/null):0
export LIBGL_ALWAYS_INDIRECT=1
```

Afterwards, disable the block rule for VcXsrv (TCP) on the public network: open *Windows Firewall* → *Advanced settings* → *Inbound Rules* → *VcXsrv windows x server* (Public, TCP) → *Disable Rule*.

Add a new rule for TCP: right click on *Inbound Rules* → *New Rule...* → *Port* → *TCP port 6000* → select defaults in the remaining windows. Open the properties of your new rule, switch to the *Scope* window, select *These IP addresses* in the *Remote IP address* window and add the address *172.16.0.0/12*.

Additional information and support can be found here and here.

Testing the X11 Forwarding

Finally, you can test X11 forwarding by installing the x11-apps and executing xeyes in the WSL:

```
$ sudo apt install x11-apps
$ xeyes
```

Remark for Linux and macOS Users

The following sections will demonstrate the installation of required software tools using the **APT** package manager, which is available on **Ubuntu** and **Debian**. If you are using a different Linux distribution or macOS you should use the appropriate package manager, e.g., **YUM** on **Fedora** or **Homebrew** on **macOS**.

Git

Git is a free and open source **distributed version control system** designed to handle everything from small to very large projects with speed and efficiency.

If you are not familiar with Git you can find some information here and here. There are also thousands of tutorials on Youtube and elsewhere on the internet.

You can install Git using the following command:

```
$ sudo apt install git
```

Afterwards configure your Git installation:

```
$ git config --global user.name "FirstName OtherNames LastName"
$ git config --global user.email "email@example.com"
$ git config --global credential.helper 'cache --timeout=3600'
$ git config --global color.ui auto
```

Now you can clone the SCVP artifacts repository:

```
$ git clone https://github.com/TUK-SCVP/SCVP.artifacts.git
```

If you want to fork the repository and push your local changes create an account on GitHub.

CMake

CMake is a widely-used tool for managing the **build process of C++ projects**. CMake is available on different platforms and is **compiler independent**. Many IDEs provide support for CMake or even use CMake for their project management by default. Alternatively the tool can be used from command line. More information is provided here.

Since the SCVP artifacts as well as all exercises are also based on CMake, you should install it using the following command:

```
$ sudo apt install cmake
```

Installing SystemC

First, check if a C++ compiler and Make is available on your system. If not you can simply install the build-essential package:

```
$ sudo apt install build-essential
```

Now download the latest SystemC and SystemC AMS sources from the official websites:

```
$ wget https://www.accellera.org/images/downloads/standards/systemc/systemc-2.3.3.tar.gz
$ wget https://www.coseda-tech.com/files/Files/Proof-of-Concepts/systemc-ams-2.3.tar.gz
```

Build and install both libraries. The following steps are shown exemplarily for the SystemC library. Extract the files, create a temporary directory inside the extracted folder and navigate into it:

```
$ tar -xzf systemc-2.3.3.tar.gz
$ mkdir systemc-2.3.3/objdir
$ cd systemc-2.3.3/objdir
```

Run the configure script specifying the directory where the library should be installed at (default is `/opt/systemc/` for SystemC and `/opt/systemc-ams/` for SystemC AMS but you can also choose different directories). Further information about the configuration options is provided in the `INSTALL` files.

```
$ ../configure --prefix=/opt/systemc/
```

Afterwards, build the library and install it (N denotes the number of build jobs, set it to the number of processor cores):

```
$ make -j N
$ sudo make install
```

As last step export environment variables for SystemC. You can also add the lines to the shell startup script (e.g. `.bashrc`) to automatically export the variables on startup.

```
$ export SYSTEMC_HOME=/opt/systemc/
$ export SYSTEMC_TARGET_ARCH=linux64
```

If you have chosen a different directory for installation adapt the first variable accordingly.

The second variable denotes the target architecture suffix. Check the library folder name inside the installation directory and set the variable accordingly. The library folder name usually starts with `lib-`.

Now repeat all steps for the SystemC AMS library. Finally, export the following environment variable (and adapt it accordingly):

```
$ export SYSTEMC_AMS_HOME=/opt/systemc-ams/
```

Testing the Installation

All required tools are now installed and you can test if your system is working. Create a build folder inside the SCVP artifacts repository and navigate into it:

```
$ mkdir SCVP.artifacts/build
$ cd SCVP.artifacts/build
```

Create a makefile using CMake and build the project:

```
$ cmake ..
$ make -j N
```

Now execute one of the examples, e.g., the `tlm_protocol_checker`:

```
$ ./tlm_protocol_checker/tlm_protocol_checker
```

You should see the following program output:

```
@40 ns Write Data:  103
@80 ns Write Data:  198
@120 ns Write Data:  105
@160 ns Write Data:  115
@200 ns Read Data:  103
@240 ns Read Data:  198
@280 ns Read Data:  105
@320 ns Read Data:  115
```

For an easier and less error-prone development you can also use an IDE to do the exercises, e.g., Visual Studio Code, CLion or Qt Creator. All IDEs support CMake for project management.

Remark for Windows Users

With WSL you can also use your Windows installation of Visual Studio Code and compile and run your code in the Linux environment using a plugin. More information can be found here and here.

Exercise 2: SystemC and Virtual Prototyping

SystemC Modules

Lukas Steiner

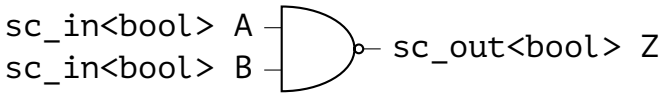
WS 2024/2025

The source code to start this execise is available here:
https://github.com/TUK-SCVP/SCVP.Exercise2

Task 1

NAND Gate

In this task you will write your first SystemC module. The module should have the name `nand` and should implement the functionality of a NAND gate, shown below.



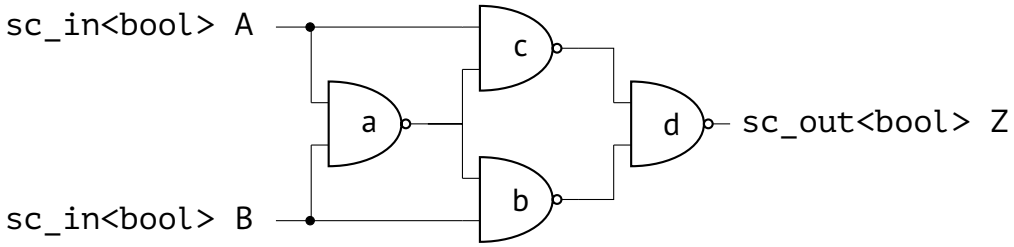
As input and output signals `sc_in` and `sc_out` should be used with the template type `bool`. The input and output signals should be initialized with a proper name in the `SC_CTOR`. The module should have one `SC_METHOD` called `do_nand()`, which is sensitive to the input signals A and B. The module should be implemented in the file `nand.h`.

In order to test your module make sure `nand_main.cpp` is included in the CMake file `CMakeLists.txt`. After successfully testing your NAND gate change the project file to include `exor_main.cpp`. This is necessary to test your next SystemC module in which you will implement an XOR gate using four instances of your NAND.

Task 2

SystemC Module Hierarchy – XOR

In this task you will write a SystemC module that is composed of other SystemC modules. The module should have the name `exor` and should implement the functionality of an XOR using only NAND gates, as shown below.



In order to connect the `nand` modules, you need additional helping signals which you will implement by using the `sc_signal<bool>` datatype. The signals should have the names `h1`, `h2` and `h3`. All input, output and helping signals as well as the `nand` modules should be initialized properly with a name from the `SC_CTOR(exor)`.

If you are done with the implementation, have a look at the `SC_MODULE`s `stim` and `mon` and the `sc_main()` function and try to understand what these components are doing.

Why is the `stim` class using an `SC_THREAD` for its process and not an `SC_METHOD`?

Now lets compile and run your program. If you did everything correctly, you should see the following output:

```
time     A    B    F
0 s      0    0    1
0 s      0    0    0
10 ns    0    1    0
10 ns    0    1    1
25 ns    1    0    1
35 ns    1    1    1
35 ns    1    1    0
45 ns    0    0    0
```

Why are you seeing several outputs for each time, sometimes even with wrong results?

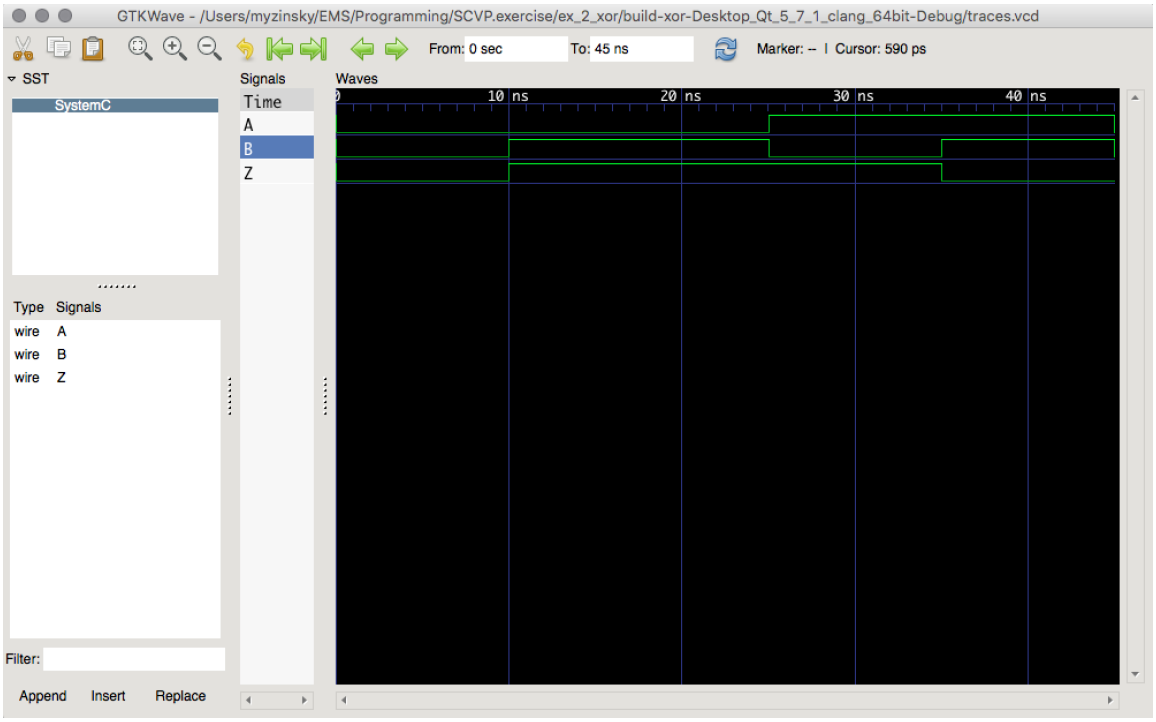
Task 3

Debugging Tracing

Additionally to the mon component, use the waveform feature of SystemC in the `sc_main` method before `sc_start`. Then use the tool GTKWave in order to have a look on the waveform. The file is located inside the build folder.

Information on this feature can be found here: https://www.doulos.com/knowhow/systemc/tutorial/debugging/

In GTKWave you have to drop the signals to the waveform and you have to zoom out a little in order to see the final result:



Task 4

Clocked Processes

Add an `sc_clock` to the `sc_main` function. Remove the `wait(XX, SC_NS)` statements in the `stim` module and replace them with empty `wait()` statements. Add an `sc_in<bool> Clk` to the `stim` and the mon components and make the processes of both modules only sensitive to the positive edge of the clock. Then connect the `sc_clock` in the `sc_main` to the modules. What you will observe at the terminal output? Now add the clock signal to the waveform and analyze it with GTKWave.

Exercise 3: SystemC and Virtual Prototyping

Exercise on State Machines

Lukas Steiner
WS 2024/2025

The source code to start this execise is available here:
`https://github.com/TUK-SCVP/SCVP.Exercise3`

Task 1

DNA Processing

Deoxyribonucleic Acid (DNA) is the building block of life. It contains the information a cell requires to synthesize protein and to replicate itself, to be short it is the storage repository for the information that is required for any cell to function. The famous double-helix structure is composed of four nucleotide bases:

- *Adenine* (A)
- *Guanine* (G)
- *Thymine* (T)
- *Cytosine* (C).

Each base has its complementary base, i.e., A will have T as its complimentary and similarly G will have C.

A DNA sequence looks something like this:

```
...ATTGCTGAAGGTGCGG...
   ||||||||||||||||
...TAACGACTTCCACGCC...
```

The previous sequence can also be written shortly as `ATTGCTGAAGGTGCGG`.

Regular Expressions (Regex)

Regular expressions, or *regexes*, are a very powerful tool for pattern matching and to handle strings and texts. This is useful in many contexts, e.g., parsing of text files or analysis of string data, i.e., searching and replacing. A powerful script language that supports regexes intensively is *Perl*. Therefore, many other programming languages adopted *Perl-Compatible Regular Expressions* (PCRE)[1]. A regular expression consists of a pattern string, which is usually surrounded by two slashes `/.../`, as shown in the following code snippet (Perl syntax):

```
my $str = "17-06-1987";

if ($str =~ /^\d\d-\d\d-\d\d\d\d$/) {
    print "Its a date\n";
}
```

The shown regex tests if the string `$str` is a valid date. If the input string is `"1-1-1"` there will not be an output. If `^` (caret) is the first character in a regex it has to match from the beginning of the string. If `$` (dollar) is the last character of the regex it has to match to the end of the string. The keyword `\d` is an abbreviation for a set of characters that matches for single digits. However, it can also be matched for normal text as seen for the `-` character. There are several other abbreviations for sets of characters available for clever matching:

- `.` : matches any character (including newline)
- `\d`: matches a digit (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
- `\D`: matches a non-digit
- `\s`: matches a whitespace character (including tabs)
- `\S`: matches a non-whitespace character
- `\w`: matches a word character
- `\W`: matches a non-word character

It is also allowed to group these sets by using the `[...]` parentheses, e.g., `[\s\d]` matches whitespaces and/or digits or `[a-z]` matches all small letters from a to z.

[1] See manpage `man pcrepattern` or *Mastering Regular Expressions* by Jeffrey E. F. Friedl.

In order to extract the data provided in the string the parentheses operator `(...)` can be utilized. Parentheses allow us to capture whatever is matched within their bounds by using the capture variables `$1, $2, ...`

```
if ($str =~ /^(\d\d)-(\d\d)-(\d\d\d\d)$/) {
    print "Day: $1   Month: $2   Year:  $3 \n";
}
```

Quantifiers can be used in order to keep the regexes compact. They denote how many characters should match in total. By default, an expression is automatically quantified by `{1,1}`, which means it should occur at least once and at most once, i.e., it should occur exactly once. In the following list E stands for expression. An expression is a character, or an abbreviation for a set of characters, or a set of characters in square brackets, or an expression in parentheses.

- `E{n}`: matches exactly `n` occurrences of E (`E{n}` is the same as repeating E `n` times. For example, `/x{5}/` is the same as `/xxxxx/`.)
- `E{n,}`: matches at least `n` occurrences of E
- `E{0,m}`: matches at most `m` occurrences of E
- `E{n,m}`: matches at least `n` and at most `m` occurrences of E

There are some special quantifiers, which are used very often:

- `E?`: matches zero or one occurrences of E (`E?` is the same as `E{0,1}`.)
- `E+`: matches one or more occurrences of E (`E+` is the same as `E{1,}`.)
- `E*`: matches zero or more occurrences of E (`E*` is the same as `E{0,}`. The `*` quantifier is often used mistakenly where + should be used).

There is also the possibility to use alternatives (like a logical or) by using the pipe operator in a parentheses `(...|...)`. The following expression will match the date `"17-06-87"` as well as the date `"17-06-1987"`:

```
$str = "17-06-87";

if ($str =~ /^(\d*)-(\d+)-(\d{2}|\d{4})$/) {
        print "Day: $1   Month: $2   Year:  $3 \n";
}
```

If the delimiter between the digits is allowed to be anything, the "." abbreviation can be used:

```
$str = "17/06/87";

if ($str =~ /^(\d+).(\d+).(\d+)$/) {
    print "Day:\t$1\nMonth:\t$2\nYear:\t$3\n";
}
```

Note that if you want to match the dot explicitly you have to use `\.`. With this method you can easily parse HTML content:

```
$str = "<h1>HTML Headline</h1>";

if ($str =~ /^<h1>(.*)<\/h1>$/) {
    print "$1\n";
}
```

Regexes can be arbitrarily complex. For example,

```
/^[a-zA-Z0-9.!#$%&'*+/=?^_`{|}~-]+@[a-zA-Z0-9-]+(?:\.[a-zA-Z0-9-]+)*$/
```

is the regex to check whether an e-mail address is valid or not. Regexes are also important to analyze DNA sequences. Recently, a special in-memory computing device has been presented by Micron, called Automata Processor[2]. This special DRAM has the ability to execute pattern matching internally. There is no need to transfer the data from memory to CPU in order to perform an analysis.

[2] Paul Dlugosch, Dave Brown, Paul Glendenning, Michael Leventhal, Harold Noyes, *An Efficient and Scalable Semiconductor Architecture for Parallel Automata Processing*, IEEE Transactions on Parallel and Distributed Systems

Implementation of Regex with an FSM

The previously introduced regexes can be implemented with state machines. Figure 1 shows a simple state machine for the regex `/GAAG/` or `/GA{2}G/`. This regex is used in order to check if the pattern GAAG is part of a DNA sequence. The state machine

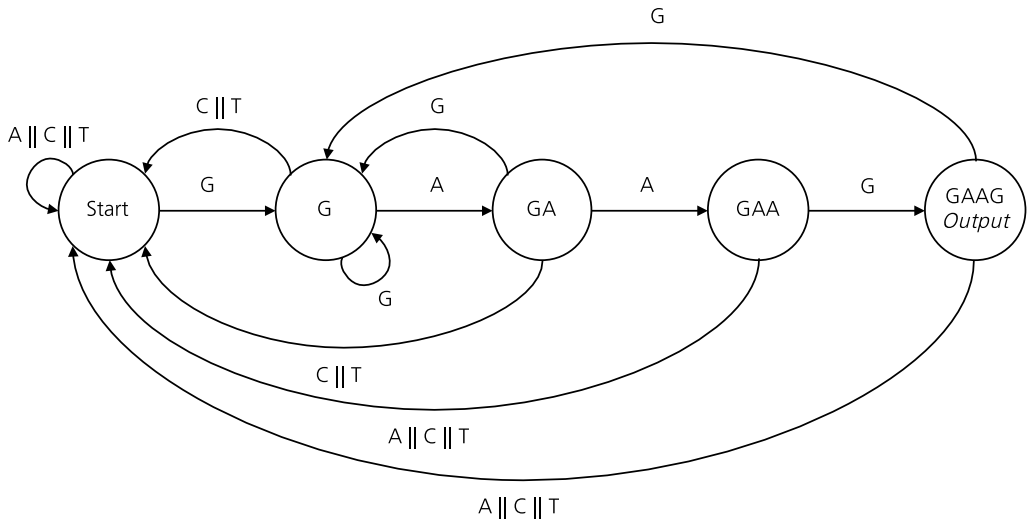


Fig. 1: Simple State Machine for the Regex `/GAAG/`

has 5 states. When the state GAAG is reached an output will be given.

First, you should implement the given state machine as an `SC_MODULE` with the class name `stateMachine`. The module should have the following inputs:

- `sc_in<char> input;`
- `sc_in<bool> clk;`

The provided module `stimuli` will provide a new DNA symbol on its output in each clock cycle. The DNA symbols are implemented with the `char` datatype. For the internal states of the state machine an enumeration should be used:

```
enum base {Start, G, GA, GAA, GAAG};
```

The module state machine should have an `SC_METHOD` called `process` that implements the behavior of the state machine (e.g., by using a `switch/case` construct). The process should not be called during the initialization phase by using the `dont_initialize()` function call in the constructor after the sensitivity list. Use

the provided testbench in order to test your program. When your FSM is in the GAAG state a printout should be done.

Second, can you estimate how often and also at which position in the string the searched pattern occurs? Implement it in your current code.

Third, draw the state machine for the `/GA{2,}+G/` regex and also modify your code accordingly.

Exercise 4: SystemC and Virtual Prototyping

Exercise on Custom `sc_interface`

Lukas Steiner
WS 2024/2025

The source code to start this execise is available here:
`https://github.com/TUK-SCVP/SCVP.Exercise4`

Task 1

Custom Interfaces, Ports and Channels

Figure 1 shows an example for a very simple *Petri Net* (PN). The goal of this exercise is to implement the semantics of PNs in SystemC by using custom interfaces, templated ports and channels. We will reach this goal going step by step through the exercise. *Transitions* will be implemented as `SC_MODULE`s and *places* will be implemented as custom channels that are connected to ports of the transitions.

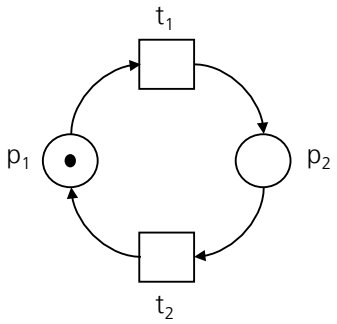


Fig. 1: Simple Petri Net

First, an interface `placeInterface` has to be implemented in `place.h`, which describes the access methods of a place channel by using pure virtual functions. The abstract class `placeInterface` inherits from `sc_interface` and should have the following pure virtual functions:

```
virtual void addTokens(unsigned int n) = 0;
virtual void removeTokens(unsigned int n) = 0;
virtual unsigned int testTokens() = 0;
```

Note: a virtual function is specified *pure* by using the *pure specifier* "`= 0`".

Second, a `place` channel should be created inside `place.h`, which inherits from `placeInterface`. The channel should have a member variable `unsigned int tokens`, which specifies the current number of tokens in the place. The initial number of tokens should be passed as a parameter to the constructor and then set in the constructor of the channel `place`.

Next the virtual functions have to be implemented:

- The method `addTokens(unsigned int n)` should add `n` tokens to the member variable `tokens`.
- The method `removeTokens(unsigned int n)` should subtract `n` tokens from the member variable `tokens`.
- The method `testTokens()` should return the value of the member variable `tokens`.

Third, an `SC_MODULE` called `transition` should be defined in `transition.h`. As usual, the constructor of the module must be defined, but there is no need to register any member function with the SystemC kernel, i.e., no need to call `SC_METHOD()` or `SC_THREAD()` for this module.

The module has two `sc_ports` from template type `placeInterface` called `in` and `out`. Furthermore, the `transition` module should have a member function `fire()` [1]. This function should check if the transition is enabled by using the method `testTokens` of the `in` port, i.e., if there exists at least one token in the place. In this case it should print

```
std::cout << this->name() << ": Fired" << std::endl;
```

remove one token from the `in` port and add a token to the `out` port. If the number of tokens in the place is zero it should print

```
std::cout << this->name() << ": NOT Fired" << std::endl;
```

[1] For the beginning of this exercise, we assume that all weights of the arcs are one and that the Petri Net has no forks or joins, i.e., a place is connected to exactly one transition's output and one transition's input, e.g., Figure 1.

In order to test our initial Petri Net implementation build an SC_MODULE called `toplevel` in `main.cpp` that reflects the PN shown in Figure 1. As a test bench the module should use the following function as an SC_THREAD.

```
void process()
{
    while (true)
    {
        wait(10, SC_NS);
        t1.fire();
        wait(10, SC_NS);
        t1.fire();
        wait(10, SC_NS);
        t2.fire();
        sc_stop();
    }
}
```

Create one instance of `toplevel` with the name `t` in the `sc_main` function. After compiling and running your code you should see the following output:

```
t.t1: Fired
t.t1: NOT Fired
t.t2: Fired
```

After firing `t1` the transaction cannot be fired again because a token is missing in the input place.

Multiports

In order to have more than one port for `in` and `out` we template the `transition` module in the following way in order to support multiple input ports:

```
template<unsigned int N = 1, unsigned int M = 1>
SC_MODULE(transition)
{
public:
    sc_port<placeInterface, N, SC_ALL_BOUND> in;
```

```
    sc_port<placeInterface, M, SC_ALL_BOUND> out;

    ...
}
```

The template parameter N denotes the number of input ports and M denotes the number of output ports, respectively.

The function `fire()` must be modified to the previous example. It should check if there is one token in each input port using the `in[i]->testTokens()` method call (where $0 \leq$ `i` $<$ N).

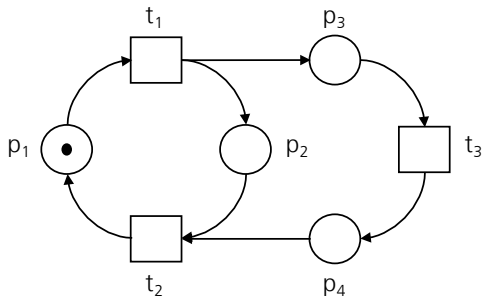


Fig. 2: Simple Petri Net

If there are enough tokens it should remove one token from each input port and add one token to each output port. The printing when firing should be done as in the previous example. In order to test our implementation, implement the PN shown in Figure 2 and use the following function as a test bench (SC_THREAD).

```
void process()
{
    while (true)
    {
        wait(10, SC_NS);
        t1.fire();
        wait(10, SC_NS);
        t2.fire();
        wait(10, SC_NS);
        t3.fire();
        wait(10, SC_NS);
        t2.fire();
```

```
        sc_stop();
    }
}
```

Note that the order of binding determines to which port number of the port array the channels are bound.

After running the application you should see the following output:

```
t.t1: Fired
t.t2: NOT Fired
t.t3: Fired
t.t2: Fired
```

Note that `t2` cannot fire until `t3` is fired.

Templated Channels

As next step we want to add weights to the channels at the input arc of a place and at the output arc of a place. First, we change the methods of the interface class as follows:

```
virtual void addTokens() = 0;
virtual void removeTokens() = 0;
virtual bool testTokens() = 0;
```

As for the `toplevel` module we use templates for the `place` channel

```
template<unsigned int Win = 1, unsigned int Wout = 1>
class place : public placeInterface
...
```

where `Win` denotes the input weight of a place and `Wout` the output weight of a place. The methods of the `place` channel have to be changed according to the new `placeInterface`:

- The method `addTokens()` should now add `Win` tokens to the member variable `tokens`.

- The method `removeTokens()` should now subtract `Wout` tokens from the member variable `tokens`.
- The method `testTokens()` should test if the number of `tokens` is greater than or equal to Wout.

The `transition` and `toplevel` modules should be adapted accordingly. By using weights of 1 the new application output should be identical to the previous one.

Modelling a Single Memory Bank

With the current code base we want to model the single memory bank example from the lecture, as shown in Figure 3. Create a `toplevel` module that reflects the memory bank; make sure to use the same names for transitions and places as shown in the picture.

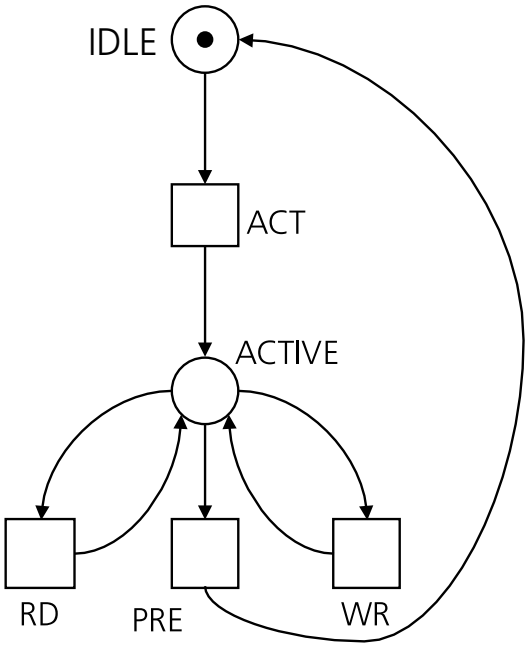


Fig. 3: Single Memory Bank Example

Test your code with the following process:

```
void process()
{
    while (true)
```

```
    {
        wait(10, SC_NS);
        ACT.fire();
        wait(10, SC_NS);
        ACT.fire();
        wait(10, SC_NS);
        RD.fire();
        wait(10, SC_NS);
        WR.fire();
        wait(10, SC_NS);
        PRE.fire();
        wait(10, SC_NS);
        ACT.fire();
        sc_stop();
    }
}
```

The application output should be the following:

```
t.ACT: Fired
t.ACT: NOT Fired
t.RD: Fired
t.WR: Fired
t.PRE: Fired
t.ACT: Fired
```

Implementation of Inhibitor Arcs

In order to implement inhibitor arcs add an additional template parameter `L` for the number of inhibitor inputs,

```
template<unsigned int N=1, unsigned int M=1, unsigned int L=0>
```

and an additional `sc_port` to the `transition` module:

```
sc_port<placeInterface, L, SC_ZERO_OR_MORE_BOUND> inhibitors;
```

In addition to the firing check for enough tokens in the input ports it is necessary to check if there are no tokens in the connected channels of all inhibitor ports by using the `testTokens()` method. In other words, the firing is only performed if there are enough tokens in all input places and no tokens in places that would inhibit it. Adjust the `fire()` method accordingly. Why did we use the `SC_ZERO_OR_MORE_BOUND` flag?

Building Hierarchical PNs

To finish the exercise create an `SC_MODULE` called `subnet` in `subnet.h` that implements the PNs in the green boxes of Figure 4[2].

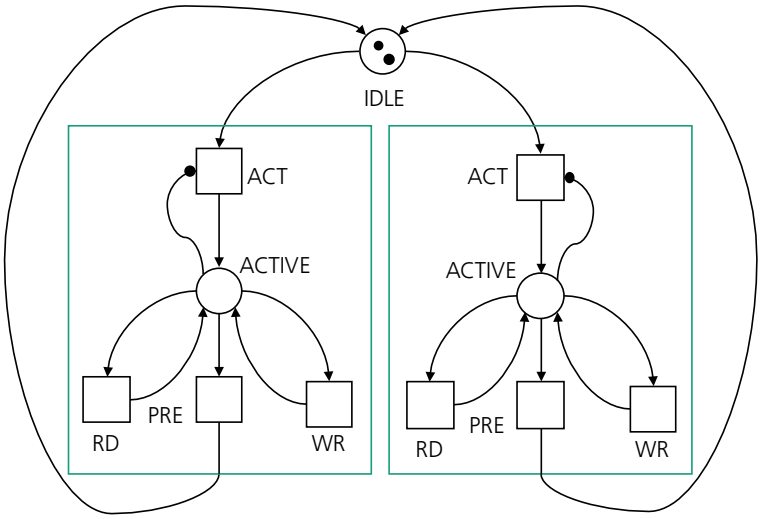


Fig. 4: Two Memory Bank Example with Subnets

Afterwards, implement the `toplevel` module in such a way that it includes two instances of `subnet`, called `s1` and `s2`. Take care that all ports are bound correctly. You can test your implementation with the following process:

```
void process()
{
    while (true)
    {
        wait(10, SC_NS);
        s1.ACT.fire();
```

[2]Hint: you can also bind input ports to input ports and output ports to output ports.

```
        wait(10, SC_NS);
        s1.ACT.fire();
        wait(10, SC_NS);
        s1.RD.fire();
        wait(10, SC_NS);
        s1.WR.fire();
        wait(10, SC_NS);
        s1.PRE.fire();
        wait(10, SC_NS);
        s1.ACT.fire();
        wait(10, SC_NS);
        s2.ACT.fire();
        wait(10, SC_NS);
        s2.ACT.fire();
        wait(10, SC_NS);
        s1.PRE.fire();
        wait(10, SC_NS);
        s2.PRE.fire();
        wait(10, SC_NS);
        sc_stop();
    }
}
```

As an output you should see the following:

```
t.s1.ACT: Fired
t.s1.ACT: NOT Fired
t.s1.RD: Fired
t.s1.WR: Fired
t.s1.PRE: Fired
t.s1.ACT: Fired
t.s2.ACT: Fired
t.s2.ACT: NOT Fired
t.s1.PRE: Fired
t.s2.PRE: Fired
```

Exercise 5: SystemC and Virtual Prototyping

Exercise on `sc_fifo`

Lukas Steiner
WS 2024/2025

The source code to start this execise is available here:
`https://github.com/TUK-SCVP/SCVP.Exercise5`

Task 1

Kahn Process Networks

Figure 1 shows an example for a *Kahn Process Network* (KPN). It consits of three processes, has zero inputs and one output e. The process **Add** reads one integer number from each of its input a and c and writes the sum of both numbers to its output b ($b = a + c$). The **Split** process copies its input b to two FIFOs (a and d) and to the output signal e. The process **Delay** writes its input d to its output c. Two FIFOs are initialized with single values: $b = 1$ and $c = 0$.

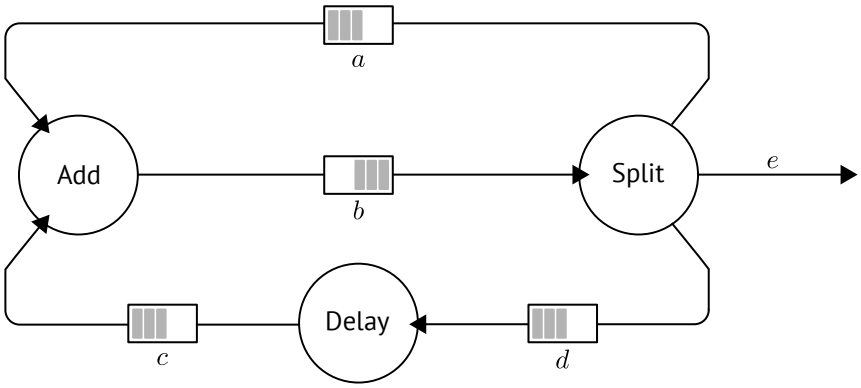


Fig. 1: Simple Kahn Process Network

The task for this exercise is to implement this KPN in SystemC using `sc_fifo<T>` and SC_THREADS for the processes. Please use a FIFO depth of 10, `unsigned int` as template type `T` and blocking `read()` and `write()` for accessing the FIFOs. The output e should be printed by the **Split** process. The **Split** process should stop the simulation after 10 prints. Initialize the FIFOs b and c in the SC_CTOR.

What is this KPN doing?

Exercise 6: SystemC and Virtual Prototyping

Exercise on TLM Loosely Timed (LT)

Lukas Steiner
WS 2024/2025

The source code to start this execise is available here:
`https://github.com/TUK-SCVP/SCVP.Exercise6`

Task 1

Single Initiator and Target

Figure 1 shows an example for a very simple TLM scenario. The code on GitHub for this exercise already provides a simple initiator called `processor`, which fakes the memory access behavior of a simple 32-bit microcontroller by replaying a trace file. As a first step you should study the code of this initiator. Do you recognize the *Regular Expressions* for parsing the input trace? In C++ we have to escape the `\` with a `\`, therefore we see for example patterns like `\\d+` instead of `\d+`.

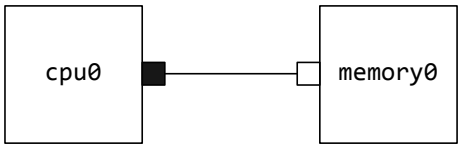


Fig. 1: Initiator and Target

As second part of this task you will implement a simple TLM-LT target called `memory` in the file `memory.h`. The memory size should be provided as a template parameter, and the standard value should be 1024 bytes. If an address greater than or equal to 1024 is accessed, a proper TLM error handling should be done (see lecture) and the simulation should be stopped with `SC_REPORT_FATAL` by the initiator. Hint: Have a look at the artifacts repository

`https://github.com/TUK-SCVP/SCVP.artifacts/tree/master/tlm_lt_initiator_target`

to see an example for an LT-based target. The functions `nb_transport_fw`, `get_direct_mem_ptr` and `transport_dbg` must be implemented as stubs, i.e.,

dummy functions as in the artifacts example. Note that in the target `wait()` should not be called. The time consumed by the memory access (20 ns) should be added to the `delay` variable of the `b_transport` function call.

In order to test your first TLM memory you have to instantiate an object of the class processor (`cpu0`) and an object of the class memory (`memory0`) like this:

```
processor cpu0("cpu0", "stimuli1.txt", sc_time(1, SC_NS));
memory<1024> memory0("memory0");
```

The second constructor parameter for the `cpu0` is the input trace to be read, and the third parameter is the inverse of the frequency, in our case 1 GHz. Then bind the initiator socket of `cpu0` to the target socket of `memory0` in the `main.cpp` file as shown in the lecture.

The `stimuli1.txt` writes some data to the memory in the beginning and reads the same data again some cycles later. If you implemented your memory properly you will see that the data is the same as it was written to the memory before.

Remember to copy the input trace files `stimuli1.txt` and `stimuli2.txt` to the same directory where your executable is located.

You might face the following error when executing your solution:

```
Error: (E549) uncaught exception: regex_error
In file: sc_except.cpp:100
In process: cpu1.process @ 0 s
```

Then uncomment the following line in `processor.h`, rebuild your project and try again.

```
#define GCC_LESS_THAN_4_9_DOES_NOT_SUPPORT_REGEX
```

Task 2

Multiple LT Initiators and Targets

Figure 2 shows an example for a more sophisticated scenario. In this case we have two processors (`cpu0` and `cpu1`), which are connected over an interconnect (`bus0`) to two different memories (`memory0` and `memory1`). The bus should forward each transaction to the right memory based on the memory map shown in Figure 2 (routing).

Implement a bus component in the `bus.h` file and update the `main.cpp` accordingly. `cpu0` should replay the trace file `stimuli1.txt` and `cpu1` the trace file `stimuli2.txt`. Each memory should have a size of 512 bytes.

Hint: Have a look at the artifacts repository

`https://github.com/TUK-SCVP/SCVP.artifacts/tree/master/tlm_lt_initiator_interconnect_target`

again to see an example for an LT-based interconnect with routing.

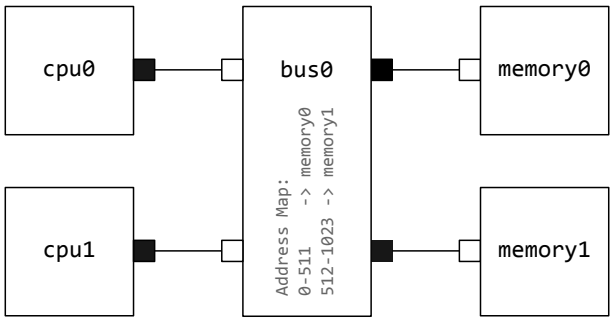


Fig. 2: Initiators, Interconnect and Target

Task 3

Temporal Decoupling

Finally, we will utilize the concept of temporal decoupling to speed up our simulation.

Change the SC_THREAD of the processor from `processTrace` to `processRandom` (inside `processor.h`). This process does not use an input trace file for memory traffic generation but internally generates random time stamps and addresses. In order to achieve a noticeable speedup with temporal decoupling we cannot use input trace files because the file parsing consumes a lot more wall-clock time than the actual system behavior simulation.

In addition, all `cout` statements are removed for a more accurate comparison.

Now modify the processor in such a way that it uses a quantum keeper for temporal decoupling.

Hint: Have a look at the artifacts repository

`https://github.com/TUK-SCVP/SCVP.artifacts/tree/master/tlm_quantum_keeper`

again to see an example for temporal decoupling.

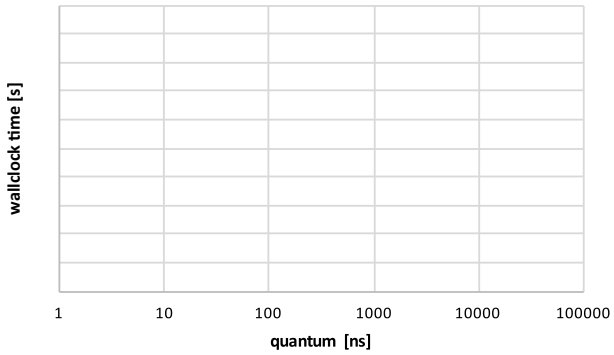


Fig. 3: Quantum vs. Wall-Clock Time

Execute the code with quantums ranging from 1 ns to 100000 ns. For measuring the wall-clock time you can use the `time` command

```
time ./exercise6
```

Put the values into the graph template in Figure 3. You should observe variations in wall-clock time of around 40%.

Exercise 7: SystemC and Virtual Prototyping

Exercise on TLM Approximately Timed (AT)

Lukas Steiner

WS 2024/2025

Task 1

TLM Approximately Timed (AT)

This exercise is different. You don't have to write code. Surprised? The reason is simple: the AT protocol is pretty complex. Therefore, in this exercise you should study several code examples in order to understand how we can model in the AT coding style. To really learn the AT style you have to do a project where these code examples could be a starting point.

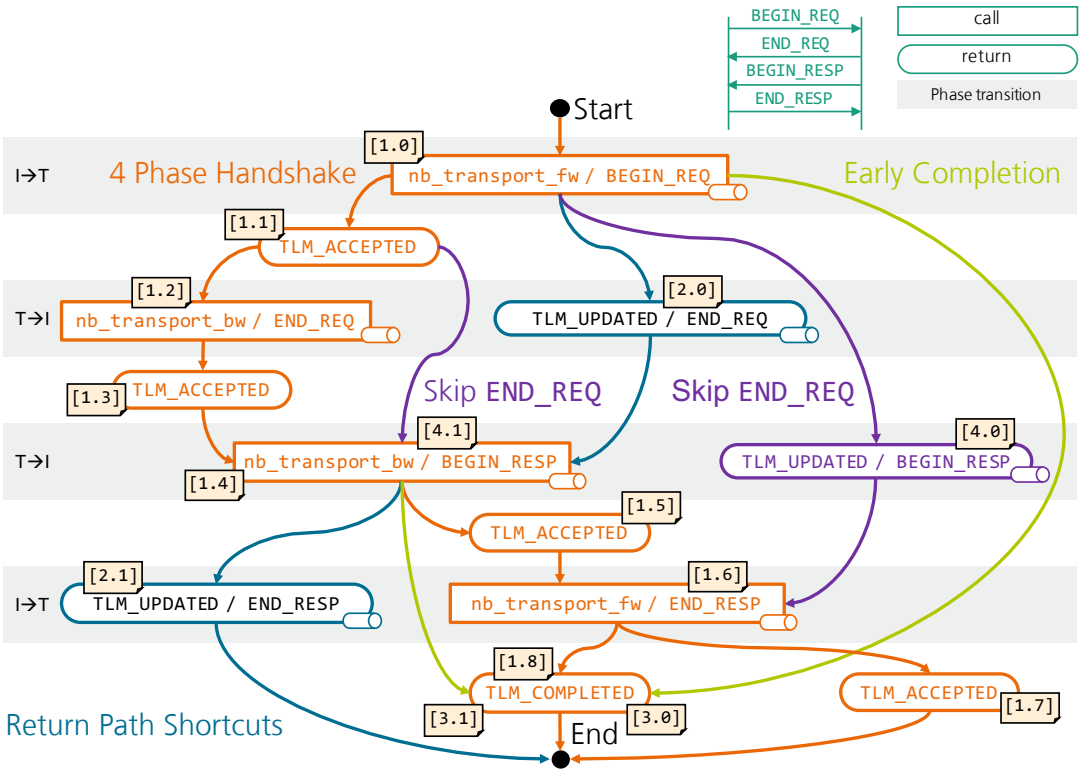


Fig. 1: The AT Cheat Sheet

Figure 1 shows a cheat sheet with the different possibilities that the TLM AT protocol offers. The annotated references `[X,Y]` are placed into the source code for a better orientation.

1. Four Phase Handshake

The code for the first example is available here:

`https://github.com/TUK-SCVP/SCVP.artifacts/tree/master/tlm_at_1`

The example consists of an AT *Initiator* and *Target*, which both follow the *Four Phase Handshake*. The initiator will randomly send reads and writes to a random address and uses this random address also as data (also note how a `reinterpret_cast` can be used in TLM). The target will negate the data in the case of a read command. The intention is to show the concept and there is no deeper meaning behind the example. The target is only able to handle one incoming transaction and uses the backpressure mechanism (deferring END_REQ) to stop the initiator from sending new requests. Later we will see how targets could handle multiple incoming transactions.

Note that in general there are several ways to implement solutions that are standard compliant, this is just one way to do it.

In order to check if programmed models are standard compliant, the example uses the *TLM2 Base Protocol Checker* from John Aynsley:

`https://github.com/TUK-SCVP/SCVP.artifacts/tree/master/tlm_protocol_checker`

This checker stops the simulation immediately if there is a non-standard-compliant behavior in the current simulation.

Please study the code of the AT initiator and target by following the four phase handshake shown in Figure 1.

2. Return Path Shortcuts

The following three examples are build on top of example 1 by inheritance from initiator or target. The code for example 2 is available here:

`https://github.com/TUK-SCVP/SCVP.artifacts/tree/master/tlm_at_2`

Please study the code by following the *Return Path Shortcuts* in Figure 1. Note that the target decides to go the `[2.0]` or the normal path `[1.0]` and the initiator decides to go the path `[2.1]` or `[1.5]`. It is also possible that only one of the two shortcuts is taken.

3. Early Completion

The code for example 3 is available here:

`https://github.com/TUK-SCVP/SCVP.artifacts/tree/master/tlm_at_3`

In this example the target ends the transaction by an early completion, which is similar to the `b_transport` of LT coding style. Please study the code by following the *Early Completion* path in Figure 1.

4. Skip END_REQ

The code for example 4 is available here:

`https://github.com/TUK-SCVP/SCVP.artifacts/tree/master/tlm_at_4`

This example shows the two ways how the END_REQ phase can be skipped. Please study the code by following the *Skip* END_REQ paths in Figure 1.

5. Backpressure

The code for this example is available here:

`https://github.com/TUK-SCVP/SCVP.artifacts/tree/master/tlm_at_backpressure`

In this example the target has a request buffer for 8 elements. When the buffer is full, the target will apply backpressure to the initiator by deferring END_REQ. Please study the code and observe the output: Transactions will happen out-of-order!

Exercise 8: SystemC and Virtual Prototyping

Exercise on TLM Payload Extensions

Lukas Steiner
WS 2024/2025

The source code to start this execise is available here:
`https://github.com/TUK-SCVP/SCVP.Exercise8`

Task 1

Setup of TLM AT Model

Setup a TLM AT simulation model of the system shown in Figure 1. The CPU and the memory models should use *Simple Sockets*. The target has an input buffer of size 8 and should implement backpressure. The bus should use *Multipassthrough Sockets* and should use *Payload Extensions* for routing.

To make things easier you can use the code that is provided on GitHub.

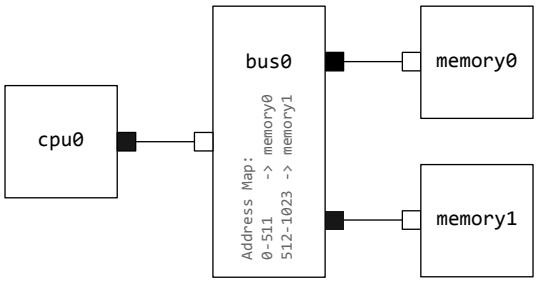


Fig. 1: Initiators Interconnect and Target

The initiators model the usual behavior of the application that we execute on them. If you execute the model you can observe how the buffers of the targets are behaving on the simulation output. By running the simulation with the following command, we can estimate the maximum number of transactions in the target buffer.

```
$ ./exercise8 | grep Buffer | awk '{print $9}' | sort | tail -n 1
```

Try to understand this bash scripting.

Task 2

Design Space Exploration of the Buffer Size

It seems that the *target buffer size* is overbuilt. The target buffer size can be reduced to 6 without any problems. Like this:

```
Target *memory1 = new Target("M1", 6);
Target *memory2 = new Target("M2", 6);
```

Please verify that the simulation time has not changed. However, the greedy management of our company forces us to save even more resources. So we have to find a trade-off between execution speed and buffer size.

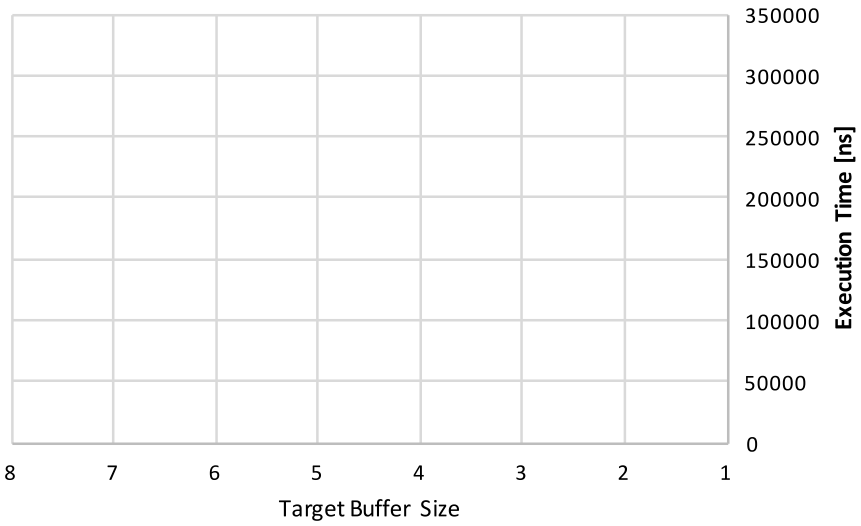


Fig. 2: Trade-off Estimation

Vary the buffer size from 8 to 1 and document the simulation time, i.e., the execution time of the CPU in Figure 2.

Which buffer size is reasonable to fulfill the request of the management?

Task 3

Wall-Clock Time and cout

Our simulation has a lot of `cout` statements. Measure the simulation time by running:

```
$ time ./exercise8
```

Now, comment out all cout statements in the simulation models and measure the wall-clock time again.

How much difference do you see?