Exercise 3: SystemC and Virtual Prototyping

Exercise on State Machines

Matthias Jung, Lukas Steiner
WS 2022/2023

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

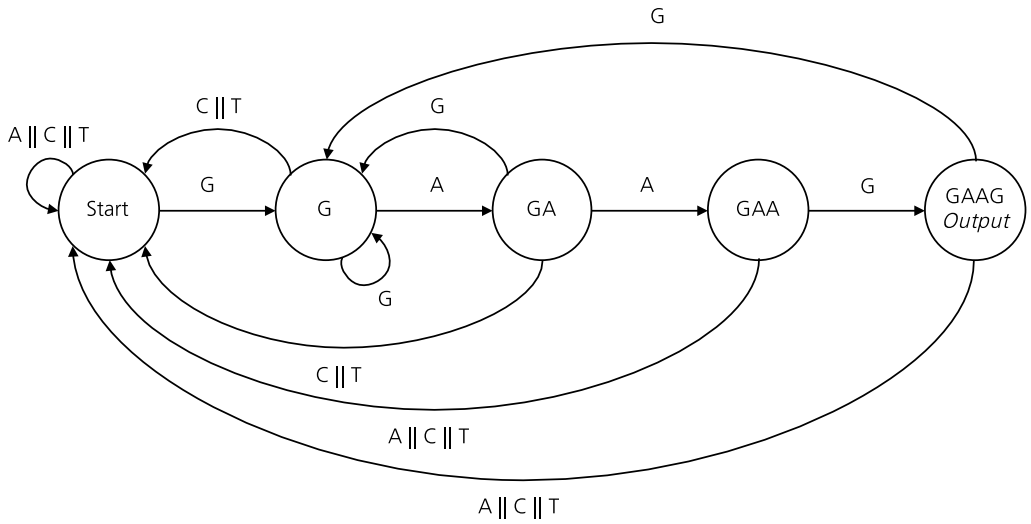

Fig. 1: Simple State Machine for the Regex `/GAAG/`

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.