Exercise 3: SystemC and Virtual Prototyping

Exercise on State Machines

Matthias Jung
WS 2017/2018

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 the life. It contains the information the 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 out 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 some thing like this:

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

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

Regular Expressions (Regex)

Regular expressions, or Regexs, 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 Regexs intensively is PERL. Therefore, many other programming language adopted *Perl-Compatible Regular Expressions* (PCRE)[1]. A regular expression consists a pattern string which is usually surrounded by two slashes `/.../`, as shown in the following code snip (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 `$str` is a valid date. If the input string would be `"1-1-1"` there won't be any output. Note that, when `^` (caret) is the first character in a regexp, it means the regexp must match from the beginning of the string and when `$` (dollar) is the last character of the regexp, it means the regexp must match to the end of the string. The keyword `\d` is an abbreviation for a sets of characters, which matches for single digits. However, it can also be matched for normal text as seen for the `-` character. There are several other abbreviations for a 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 for example `[\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 use the data provided in the string the parentheses operator `(...)`. Using parentheses allows 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 Regexs compact. They denote how many characters should match exactly. 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, `/x5/` 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. It is the same as `E{0,}`
  (The * quantifier is often used in error 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 on the date `"17-06-87"` as well to 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 can 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";
}
```

Regexs can be arbitrarily complex. For Example,

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

is the Regexs to check whether an Email is correct or not. Regexs are important to analyze e.g. DNA sequences. Recently, a special in-memory computing device has been presented by Micron, called Automata Processor[2]. This special DRAM memory has the ability to execute pattern matching directly internally. There is no need to transfer the data from the memory to the CPU in order to do this kind of 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 a FSM

The previously introduces Regexs can be implemented by 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 an DNA sequence. The state

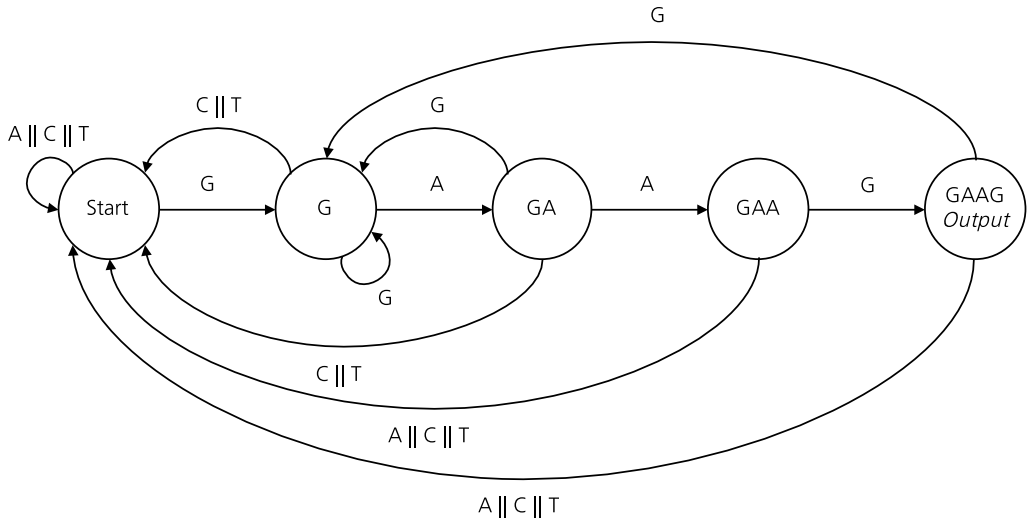

Fig. 1: Simple State Machine for the Regex `/GAAG/`

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

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

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

The already implemented module `stimuli` will provide a new DNA symbol each clock cycle on its output. 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` which 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 already implemented 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 here and modify also your code: