02135 Assignment 1:
Implementation of an FSMD simulator in Python

Luca Pezzarossa [lpez@dtu.dk]

February 4th, 2022

1 Introduction

This document describes the first assignment of the course 02135 - Introduction to Cyber Systems. In this assignment, you are required to implement a simulator for Finite State Machines with Datapath (FSMDs) in Python. The FSMDs computational model is a mathematical abstraction largely used in the design of digital logic and computer programs. Therefore, by developing a simulator, the assignment aims to give you a detailed understanding of the FSMDs computational model.

This assignment should be carried out in **groups of three people** and, in addition to the implementation of the FSMD simulator, you are also required to prepare a short report describing the approach used in its implementation (further details are provided in Section 5). All the material related to this assignment can be found on the DTU-Learn course page at the following location:

`DTU-Learn/Course content/Content/Assignments/Assignment 1`

The deadline for this assignment is **Sunday 27th February 2022 at 23:59**. By this date, you have to hand-in an electronic version of the short report in PDF format and the source files as ZIP archive using the assignment utility in the DTU-Learn course page at the following location:

`DTU-Learn/Course content/Assignments/Assignment 1`

This document is divided into 4 sections and 1 appendix:

- Section 2 provides a general FSMD background and links to the related literature.
- Section 3 describes the simulator specifications, the format of the files used as simulator input, and the additional code we provide to facilitate the simulator implementation.

- Section 4 describes the tests used to prove the correct functionality of the implemented simulator.
- Section 5 lists the assignment tasks and the report requirements.
- The appendix describes the provided python code.

There are many ways to implement an FSMD simulator. As we are not enforcing any particular approach (with the exception of the format of the files provided as simulator input), it may be difficult for the teachers to understand your implementation and thus, help to debug your code by taking into account the following hints:

- Structure your code such that it is clear what the different parts are doing.
- Think about testing your code while planning.
- Use a structural approach in the implementation.
- Comment your code.
- Deliver all the source code needed to test the simulator and, if necessary, a README file with instructions on how to run it.

Feel free to ask or send an e-mail to the assignment author if you have questions regarding practical matters and the assignment in general.

2 FSMD background

The FSMD computational model is a mathematical abstraction that describes a computing systems behavior and functionality as a controller (finite-state machine), which controls the program flow, and a datapath, which performs data processing operations. In Chapter 8 of the textbook [1] you can find a formal description and examples of the FSMD model. Further description can be found in the initial part of [2] (up to page 3) where a very clear formal description of an FSMD is provided. We recommend that you read this material before starting to implement the simulator.

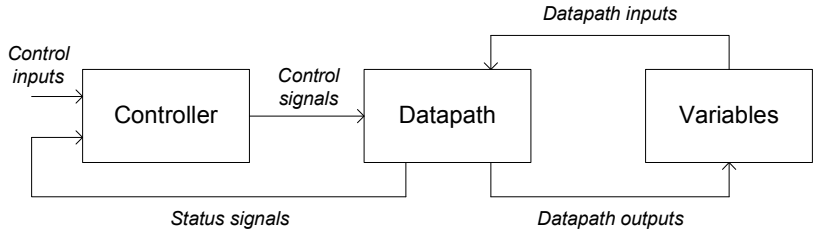

Figure 1: Block diagram of an FSMD.

Figure 1 shows a block diagram of the FSMD structure that we address in this assignment. The controller is a finite state machine that can only be in one state

at a given time. Depending on the input signal from the external environment and by the status signals from the datapath, the controller makes the decision about the transition to the next state and generates the control signal for the datapath. The datapath performs operations on a set of variables stored in memory depending on the control signals. The datapath also generates the status signals, by executing comparison operations in the variables and returning Boolean values.

An FSMD can be described using an oriented graph representation or with a transition table. In this assignment, we use the transition table approach, since it can easily be used as input argument for the simulator. An example of a transition table can be found on page 3 of [2]. In the example, for each *present state*, the table lists the *next state* and the *output signals* depending on which of the *input conditions* is evaluated *true*. The *input condition* is a Boolean expression and only one *input condition* can be *true* in any given state. The *output signals* correspond to a list of operations performed by the datapath. A list of operations is also called 'instruction'. When a transition is independent to the input, the *input condition* is set to be always *true* (also indicated as '1'). When no operations are performed in a given state, the *output signals* are *NOP* (also indicated as '-').

3 Simulator specifications

In this section, we provide the simulator specifications and we describe the format of the files used as simulator input and the expected output. We also give an overview on some additional code we provide to facilitate the simulator implementation.

3.1 Overview

The simulator should be implemented in Python 3 and named `fsmd-sim.py`. It should be cycle based, which means that in each cycle a single transition of the FSMD is performed. A cycle counter should keep track of the progression of time.

The simulator should accept the following arguments:

```
python3 fsmd-sim.py <max_cycles> <description_file>
                                                <stimuli_file>
```

The `<max_cycles>` is an integer number that specifies for how many cycles the simulator should run. Therefore, the simulator terminates when the cycle counter reaches the value specified as `<max_cycles>`. The `<description_file>` and the `<stimuli_file>` respectively provide the description of the FSMD and of the environment around it. The `<stimuli_file>` is optional and can be

omitted if the FSMD does not have any input from the external environment. The structure of these two files is described in Subsection 3.2.

For example, to simulate a maximum of 100 cycles using the files `gcd_desc.xml` and `gcd_stim.xml` located in the `test_2` folder, the command is (for a command line shell):

```
python3 fsmd-sim.py 100 ./test_2/gcd_desc.xml
                                    ./test_2/gcd_stim.xml
```

If you use an IDE, such as PyCharm, you can edit the command arguments clicking on 'Select Run/Debug Configurations' and going to the 'Edit Configurations' (this may be slightly different depending on the IDE version). Note: in some computers, after installing Python, you may need to add the path to the Python executables to the PATH environment variable in order to be able to execute the Python command from the shell.

The simulator should have a textual output in the Python shell. The minimum that the simulator should report is the state of the variables at the end of the simulation. However, for completeness and for debug reason we recommend that that simulator outputs information for each cycle so that you can follow the complete simulation step-by-step.

As previously mentioned, we do not enforce any particular approach, with the exception of the format of the files provided as simulator input. Therefore, you are free to make your own design decisions. We provide some code to facilitate the simulator implementation as described in Subsection 3.3. Also in this case, you are free to use the provided code or to develop your own simulator from scratch.

3.2 Input files structure

The simulator takes in input two XML files: a FSMD description file and a FSMD stimuli file. XML is a largely used markup language in describing tables and graphs. In the following, we describe the structure of these files.

Please note that since the characters `&`, `<`, `>`, `"`, and `'` are reserved in XML, they need to be escaped as follows:

- Ampersand (`&`) is escaped to `&` .
- Less than (`<`) is escaped to `<` .
- Greater than (`>`) is escaped to `>` .
- Double quotes (`"`) are escaped to `&quot;` .
- Single quotes (`'`) are escaped to `&apos;` .

FSMD description file

The FSMD description file contains the description of the FSMD. The root tag of the FSMD description file is `<fsmddescription> </fsmddescription>`. This marks the beginning and the end of the FSMD description. In the following, we explain the structure of this file using the `test_2/gcd_desc.xml` file as example. The symbol '`...`' means that a section of code is omitted.

At the beginning we find the list of all the states of the controller:

```
<statelist>
  <state>INITIALIZE</state>
  <state>TEST</state>
  ...
  <state>FINISH</state>
</statelist>
```

This is followed by the declaration of the initial state:

```
<initialstate>INITIALIZE</initialstate>
```

We then find the list of the inputs (from the external environment) and the list of the variables used by the datapath.

```
<inputlist>
  <input>in_A</input>
  <input>in_B</input>
</inputlist>

<variablelist>
  <variable>var_A</variable>
  <variable>var_B</variable>
</variablelist>
```

The operation list lists all the operations that the datapath can perform. Each operation has a name and an expression. The expression must be a Python compatible expression (so that it can be evaluated with `eval()`) that uses variables and inputs names as operands.

```
<operationlist>
  <operation>
    <name>init_A</name>
    <expression>var_A = in_A</expression>
  </operation>
  ...
  <operation>
    <name>A_minus_B</name>
    <expression>var_A = var_A - var_B</expression>
  </operation>
  ...
</operationlist>
```

The condition list lists all the conditional tests that the datapath can perform. Also in this case, each condition has a name and an expression. The expression must be a Python compatible conditional statements (also evaluated with `eval()`) that uses the variables as operands.

```
<conditionlist>
  <condition>
    <name>A_equal_B</name>
    <expression>var_A == var_B</expression>
  </condition>
  ...
  <condition>
    <name>B_greater_A</name>
    <expression>var_A &lt; var_B</expression>
  </condition>
</conditionlist>
```

Finally, there is the description of the transition table. For each state, a list of transitions is provided, and for each transition, the condition, instruction, and next state are listed. The condition is a Python compatible Boolean expression that uses the names of the conditions defined in the condition list. The instruction is a list of operations names defined in the operation list separated by spaces. Please note that *True* is used as a condition when a transition is independent by the input and *NOP* is used when no operations are performed.

```
<fsmd>
  <INITIALIZE>
    <transition>
      <condition>True</condition>
      <instruction>init_A init_B</instruction>
      <nextstate>TEST</nextstate>
    </transition>
  </INITIALIZE>
  <TEST>
    <transition>
      <condition>A_equal_B</condition>
      <instruction>NOP</instruction>
      <nextstate>FINISH</nextstate>
    </transition>
    ...
    <transition>
      <condition>B_greater_A</condition>
      <instruction>NOP</instruction>
      <nextstate>BMINA</nextstate>
    </transition>
  </TEST>
  <AMINB>
    ...
  </AMINB>
  ...
</fsmd>
```

FSMD stimuli file

The FSMD stimuli file describes the environment around the FSMD and gives directives to the simulator. The root tag of the FSMD stimuli file is `<fsmdstimuli> </fsmdstimuli>`. This marks the beginning and the end of the stimuli description. In the following, we explain the structure of this file using the `test_2/gcd_stim.xml` file as example.

The set-input directive specifies a certain value that an input of the FSMD should acquire at a certain moment in time. Each set-input directive specifies the cycle when the directive takes effect and the Python compatible expression to be executed. The effect of a directive is maintained until another directive affects the same input.

```
<setinput>
  <cycle>0</cycle>
  <expression>in_A = 100</expression>
</setinput>

<setinput>
  <cycle>0</cycle>
  <expression>in_B = 12</expression>
</setinput>
```

The end-state directive stops the simulator when a specified state is reached. Therefore, the simulator terminates if the end-state is reached or if the cycle counter reaches the maximum value specified as one of the simulator arguments.

```
<endstate>FINISH</endstate>
```

3.3 Provided code

In order to facilitate the implementation of the FSMD simulation, we provide a template for the `fsmd-sim.py` that includes some code for the XML files parsing using the `xmltodict` Python module (you may need to install this module with `pip` or `pip3`). The parsing code provides you with a set of variables (strings, lists, and dictionaries) that contains all the information contained in the XML files and as arguments. The code also prints to the Python shell all the parsed information. Moreover, we also provide a set of functions for executing FSMD instructions, FSMD operations, and evaluate FSMD conditions. These functions are based on the `eval()` function of Python and are described in the appendix. Feel free to use the provided code or implement your own.

The provided code is documented in the Appendix. In addition, brief comments are provided with the code itself. Please note that no checks are performed on the information parsed from the XML. Therefore, there must not be mistakes in the FSMD description and stimuli files.

4 Testing the simulator

In order to test that your simulator is functional, you should perform the three test described in this section. The first two test are provided by us and the third test should be developed by you.

4.1 Test 1

This is a simple test of a Mealy FSMD that has no interaction with the external environment. Therefore, only an FSMD description file is provided. Figure 2 shows the graph representation of the FSMD used in this test. The file for this test can be found in the `test_1` folder.

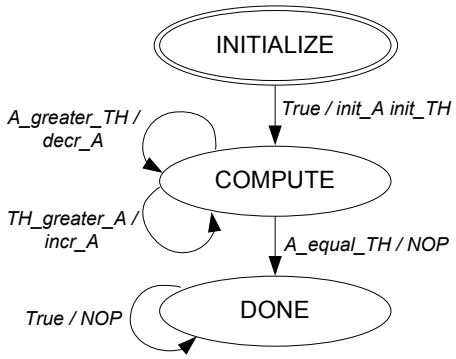

Figure 2: Mealy FSMD used in Test 1.

The FSMD works on two variables, `var_A` and `var_TH`. After initialization, in each cycle, `var_A` is decremented by 1 if greater than the threshold `var_TH` and incremented by 1 if smaller than the threshold `var_TH`. When `var_A` is equal to `var_TH`, the FSMD terminates its execution. This simple FSMD is ideal to test the basic functionality of your simulator during development and debugging.

4.2 Test 2

In this test, a Moore style FSMD calculates the greatest common divisor between two numbers provided as input during initialization. Figure 3 shows the graph representation of the FSMD used in this test. The files for this test can be found in the `test_2` folder.

The FSMD has 2 inputs, `in_A` and `in_B`, and two variables, `var_A` and `var_B`. When the execution is complete, both the variables contain the calculated greatest common divisor. The initial values are provided through the two inputs in the stimuli file, where also the end-state used to stop the simulation is defined.

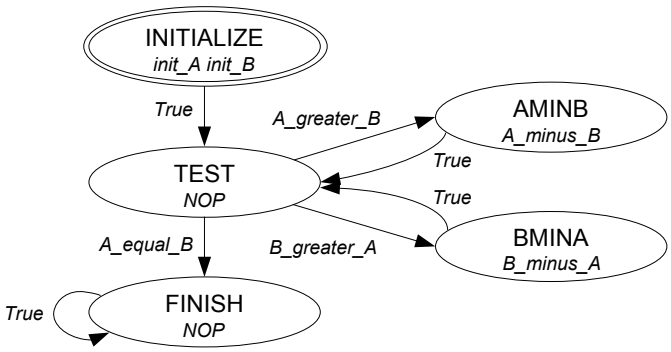

Figure 3: Moore FSMD used in Test 2.

4.3 Test 3

For this test, you are required to develop a simple FSMD and its environment. The FSMD should perform a simple computational task similar to the ones presented in Test 1 and Test 2. Test your simulator with the FSMD that you have developed.

5 Assignment tasks and report requirements

In the following, we list the task that you should perform in this assignment and the requirements for the short report. The simulator will be tested as described in Section 4. Therefore, remember to hand-in all the source code needed to test the simulator and, if necessary, a README file with instructions on how to run the simulator.

These are the tasks you should perform in this assignment:

1. Read Section 2 and the addition material [1, 2] in order acquire a clear understanding of the FSMD computational model.
2. Read carefully the simulator specifications provided in Section 3 and take a look at all the provided source files, including the tests.
3. Design and implement your simulator.
4. Test the simulator with the provided FSMD descriptions of Test 1 and Test 2 (described in Section 4).
5. Design your own FSMD (a simple one!) and test the simulator with it.
6. Prepare a short report according to the instruction provided in the following.

The short report should not be longer than 6 pages (everything included) and should provide an overall description of your simulator focusing on the main

implementation ideas and decisions. We do not need a detailed description of each aspect of the simulator. There are no specific requirements on the template to be used for the short report. Do not include the full code in the report. You can include some code snippets if these are relevant to explain certain aspects of the implementation.

References

[1] F. Vahid and T. Givargis, *Embedded System Design: A Unified Hardware/-Software Introduction*. John Wiley & Sons, Inc., 2002.

[2] A. Sudnitson, "Finite state machines with datapath partitioning for low power synthesis," June 2014.

Appendix: Documentation for the provided Python code

Variables

As previously mentioned, we provide a template for the `fsmd-sim.py`. This template includes code for the parsing of the XML files and command line arguments (i.e., reading the XML file and populating the variables that can be used by the programmer). This set of variables is described below:

• **`iterations`** : Variable of type *integer* that contains the first argument passed to the simulator, which is the maximum amount of iterations (steps/cycles) the simulator should perform.

• **`states`** : Variable of type *list* that contains the list of all states names as specified in the XML file.

• **`initial_state`** : Variable of type *string* that contains the initial-state name as specified in the XML file.

• **`inputs`** : Variable of type *dictionary* that contains the list of all inputs names and value. The names are the one specified in the XML file. The value is the actual value of a certain input at any moment in time. The default value is 0.

• **`variables`** : Variable of type *dictionary* that contains the list of all variables names and value. The names are the one specified in the XML file. The value is the actual value of that variable at any moment in time. The default value is 0.

• **`operations`** : Variable of type *dictionary* that contains the list of all the defined operations names and expressions as specified in the XML file.

• **`conditions`** : Variable of type *dictionary* that contains the list of all the defined conditions names and expressions as specified in the XML file.

• **`fsmd`** : Variable of type *dictionary* that contains the list of dictionaries, one per state, with the fields 'condition', 'instruction', and 'nextstate'. It describes the FSMD transition table. For example, the following snippet of code shows how to access (for printing for each transition) the condition, the instruction, and nextstate of a certain state 'STATE_A' (the state name). N.B. The state name does not need to be an hard-coded string, but it can be obtained by the *states* variable.

```
for transition in fsmd[STATE_A]:
  print('Condition: ' + transition['condition'])
  print('Instruction: ' + transition['instruction'])
  print('Next state: ' + transition['nextstate'])
```

• **`fsmd_stim`** : Variable of type *dictionary* that contains the list of dictionaries related to the XML stimuli file. According to this file, the fields are 'fsmdstimu-

lus', 'setinput' or 'endstate', and 'cycle' or 'expression'. Examples of its use are provided later in this appendix in the Code snippets subsection.

Functions

In addition to the set of variables, the provided code also define some functions that can be used by the programmer to develop the simulator. This functions are described below:

• **`execute_setinput(operation)`** : This function executes a Python-compatible operation passed as string on the operands stored in the dictionary 'inputs'. It is used to set the inputs. After executing this function, the value of the affected inputs in the dictionary 'inputs' changes. An example of its use is provided later in this appendix in the Code snippets subsection.

• **`execute_operation(operation)`** : This function executes a Python-compatible operation passed as string on the operands stored in the dictionaries 'variables' and 'inputs'. After executing this function, the value of the affected inputs and variables in the dictionaries 'variables' and 'inputs' changes. N.B. This is a support function used by the `execute_instruction(instruction)`, which is the one we recommend using in your implementation.

• **`execute_instruction(instruction)`** : This function executes a list of operations passed as string and spaced by a single space using the expression defined in the dictionary 'operations'. This function executes the function `execute_operation(operation)` for each specified operation. An example of its use is shown in the following snippet of code. This executes all the operations listed in the instruction field of the state 'STATE_A' as specified in the XML file. This is just an example, in your simulator you need to execute only one instruction per cycle (the one that the condition evaluates true).

```
for transition in fsmd[STATE_A]:
  execute_instruction(transition['instruction'])
```

• **`evaluate_condition(condition)`** : This function evaluates a Python-compatible boolean expressions of conditions passed as string. It uses the conditions defined in the variable 'conditions' and the operands stored in the dictionaries 'variables' and 'inputs'. Depending on the condition and on the operand, it returns True or False. An example of its use is shown in the following snippet of code. This evaluates the all the condition specified in the XML file for the state 'STATE_A'. Since, the function return True or False, you can use it in an if statement.

```
for transition in fsmd[STATE_A]:
  evaluate_condition(transition['condition'])
```

• **`merge_dicts(*dict_args)`** : A function that merges two dictionaries. N.B. This is a support function used by the `execute_operation(operation)` and `evaluate_condition(condition)`.

Code snippets

In the following, we provide a couple of snippets showing how to use the setinput function and the 'fsmd_stim' variable. The following code snippet can be used to update the values of the inputs variable according to the stimuli file content. Here you can see how the 'fsmd_stim' variable is used to access 'setinput' (in the for-loop), 'cycle', and 'expression'.

```
try:
  if (not(fsmd_stim['fsmdstimulus']['setinput'] is None)):
    for setinput in fsmd_stim['fsmdstimulus']['setinput']:
      if type(setinput) is str:
        #Only one element
        if int(fsmd_stim['fsmdstimulus']['setinput']
                                          ['cycle']) == cycle:
          execute_setinput(fsmd_stim['fsmdstimulus']
                                  ['setinput']['expression'])
          break
      else:
        #More than 1 element
        if int(setinput['cycle']) == cycle:
          execute_setinput(setinput['expression'])
except:
  pass
```

The following code snippet can be used to used to check the end-state value according to the stimuli file content. Here you can see how the 'fsmd_stim' variable is used to access the 'endstate'.

```
try:
    if (not(fsmd_stim['fsmdstimulus']['endstate'] is None)):
        if state == fsmd_stim['fsmdstimulus']['endstate']:
            print('End-state reached.')
            repeat = False
except:
    pass
```