FSMD Simulator

Implementation & Verification

Assignment 1 asks us to implement an FSMD (Finite State Machine with Datapath) simulator in Python that executes FSMDs described in XML and verifies correctness by running along three different test cases. The FSMDs provide a mathematical abstraction for modeling computation in digital systems and embedded programs. This model is separating the control flow (controller/FSM) from data processing (data path containing variables, operations and conditions)

An FSMD is an abstract model that is mainly used to describe a digital system or program as a combination of a control part (finite state machine) and a data part (variables and operations). This model is very useful as it looks quite similar to both hardware (controller and data path) and to structured programs (states and operations on variables), so the same description can guide both the hardware and the software design.

Background

In an FSMD we split the system into controllers and data paths, where the controller is a finite state machine that holds the current state and decides which transition has the right be next in the queue in a list of all conditions, on the other hand the data path contains the variables, arithmetic and logical operations, and conditions (comparisons) defined on these variables and sometimes as an option, also input signals. Each transition in an FSMD has three parts: A condition, an instruction (operations ≥ 1) and a next state. In every state, exactly one transition condition is intended to be true at a time, so the machine can take one transition per clock cycle.

Test 1 is a small example FSMD with only three parts/states: INITIALIZE, COMPUTE and DONE. When in INITIALIZE, the machine sets var_A and var_TH, then will always proceed to COMPUTE where it either increases or decreases var_A until it is equal to var_TH (var_A==var_TH) and then it moves to DONE, where it stays forever. This matches the state diagram style seen in the FSMD GCD example in the lecture slides, but in simpler form.

Simulator Architecture

Organization & Input Parsing

The simulator is implemented as a single python file fsmd-sim.py and is called as:

```
python3 fsmd-sim.py <number_of_cycles> <description_file> [stimuli_file_optional]
```

It will first of all check the number of the command line arguments and exit if they are too few or too many. It reads the FSMD description XML file using the xmltodict library that is able to make the conversion of XML elements into nested Python dictionaries. If a stimuli file is provided (optional), it will also parse that into another dictionary names fsmd_stim and if it isn't provided fsmd_stim is just an empty dictionary. After parsing, the simulator builds internal data structures from the XML content: 1)states (list of state names from the statelist), 2)initial_state (starting state id), 3)inputs (dictionary of mapping variable names to integer values, start as 0), 4)variables (same as before but for variables), 5) operations (mapping condition names to python expression strings), 6) conditions (mapping condition names to Boolean expressions), 7)fsmd (nested dictionary where each state maps to a list of transitions)

Execution Helper Functions

The simulator uses several helper functions to keep the main loop simple:

- ➔ execute_setinput: updates input signal according to expressions from the stimuli file. It takes a string like “in_A=100”, splits it at the equals sign, and evaluates the right side using python’s eval function with the current inputs dictionary as the environment.
- ➔ execute_operation: works similarly but operates on variables instead of inputs. It takes an operation expression, splits it into target and expression parts, evaluates the expression using both variables and inputs, and writes the result back to the target variable.
- ➔ execute_instruction: handles complete instructions which may contain multiple operation names separated by spaces, or the special instruction NOP (no operation). When NOP, it will return immediately, otherwise it splits the instruction into operation names and executes each one in sequence.
- ➔ evaluate_condition: determines whether a condition is true in the current cycle. For simple constants like TRUE or FALSE or 1 or 0 it returns the matching boolean. For named conditions, it replaces each condition names with its expression from the conditions dictionary, the evaluates the resulting Boolean expression.

Main Cycle-Based Simulation Loop

The simulator initializes with cycle = 0 and state = initial_state, the it will enter the main loop and in each cycle it will:

- ➔ Print the current cycle number, state, variables and inputs
- ➔ Apply any setinput operations from the stimuli file for this cycle
- ➔ Scan the transitions for the current state in order
- ➔ Evaluate each transition’s condition until one is found to be true
- ➔ Execute that transition’s instruction and updates the state
- ➔ Checks if the optional endstate has been reached
- ➔ Increments the cycle counter

The loop continues until either the maximum cycle count is reached, the endstate is reached, or no valid transition is found. At the end, it will print the total cycles executed and final variable values.

Test Cases

Test 1 - Verification of Basic FSMD Functionality

The purpose of test 1 is to examine the overall functionality of the machine by using a FSMD with a Mearly style. In this test three states are defined which are INITILIZE, COMPUTE and DONE. There are two internal variables var_A and var_TH and there are not any external inputs. This ensures that the internal state transitions and operations are tested thoroughly.

As mentioned above there are 3 states for this particular test and we begin by looking at the INITIALIZE state. In this state there are two operations in every cycle which are init_A and init_H. Init_A assigns the integer 10 to Var_A and init_H assigns the integer 0 to Var_TH. Whenever the machine is in this state the

transition to the COMPUTE state happens every time since the boolean value of TRUE is assigned to the transition condition.
The next state is the COMPUTE state which as we mentioned we transition to unconditionally from the INITIALIZE state. In this state there are three conditional branches that are executed and each will give a different outcome for the machine.

These transitions are listed below:

- ➔ A_equal_TH: if var_A == var_TH, the instruction NOP is executed and the machine transitions to DONE.
- ➔ A_greater_TH: if var_A > var_TH, the operation decr_A is executed and the machine remains in COMPUTE.
- ➔ TH_greater_A: if var_A < var_TH, the operation incr_A is executed and the machine remains in COMPUTE.

By having the structure above for the conditional branching, It ensures that the machine behaves in a deterministic way meaning that no more than one condition evaluates to TRUE in each cycle. The value of var_A is constantly decremented until it reaches zero, that is it becomes equal to var_H. Whenever var_H == var_A happens the machine finally leaves the COMPUTE state to the DONE state. Finally, in the DONE state the computation is halted while making sure that there is stable state behaviour and this is done because this state has a loop with condition TRUE and the instruction NOP. Overall, in test 1 we are only testing the logic of the execution of the FSMD since there are no external stimuli and these external stimuli are tested later on with test 2.

Test 2 - GCD Computation Using a Moore-Style FSMD with External Stimuli

The purpose of test 2 is to put the simulator in a more complex situation by applying the Moore style FSMD. The algorithm that is implemented in this test is the famous GCD algorithm that computes the greatest common divisor of two given numbers. In this test, there is an external stimuli so the process of handling the stimuli is also examined.

In this test there are 5 states INITIALIAZE, TEST, AMINB, BMINA, and FINISH. In addition to these states there are the external inputs namely in_A and in_B along with the internal inputs var_A and var_N. The goal for the gcd_stim.xml file to assign 100 and 12 to in_A and in_B respectively. This file is also the end state (FINISH) for the machine that halts the computation.

The state we begin explaining is the INITIALIZE state. In this state the input values are copied by init_A and init_B and this shows us that the datapath and the external environment are in sync together. The effect of the stimuli cann be seen in the first cycle where it resets all the inputs to 0. This shows that the simulator is working correctly by preserving the internal variables that were assigned before and at the same time updating the inputs.

The next state that this test uses is the TEST state which does not perform any arithmetic operations. This test behaves in a deterministic way by doing conditional branching and the purpose of it is to choose the next state based on the values of A and B namely:

- ➔ if A == B: the machine moves to FINISH.
- ➔ Else if A > B: the machine moves to AMINB.
- ➔ Else: the machine moves to BMINA.

In AMINB and BMINA states the following are executed:

- ➔ AMINB: executes A_minus_B, updating var_A = var_A - var_B, then returns to TEST.
- ➔ BMINA: executes B_minus_A, updating var_B = var_B - var_A, then returns to TEST.

This loop continues while (var_A != var_B) after which the transition to the FINISH state in which the computation of GCD is finished and now our variables var_A and var_B contain the computed GCD value. Test 2 confirms that the model works correctly when there is a stimuli which means the simulator is now able to handle all sorts of programs involving FSMD.

Test 3 - Custom FSMD with Input Swap

Test 3 implements a custom FSMD designed to test input manipulation and variable convergence. The FSMD has three states: INITIALIZE, COMPUTE, and DONE.

The key feature of this test is that during initialization, the input values are swapped: in_B is assigned to var_A and in_A is assigned to var_B. This is accomplished through the operations init_A (var_A = in_B) and init_TH (var_B = in_A).

The COMPUTE state then uses conditional logic similar to Test 1:

- ➔ if var_A == var_B: transition to DONE with NOP
- ➔ else if var_A > var_B: execute decr_A (var_A = var_A - 1) and stay in COMPUTE
- ➔ else if var_B > var_A: execute incr_A (var_B = var_B + 1) and stay in COMPUTE

The stimuli file provides inputs in_A = 1000 and in_B = 100 at cycle 0, then resets them to 0 at cycle 1. The endstate is defined as DONE.

After the swap during INITIALIZE, var_A = 100 and var_B = 1000. In COMPUTE, since var_B > var_A, the condition B_greater_A is true and operation incr_A executes, incrementing var_B. This causes var_B to increase each cycle, moving it away from var_A rather than toward convergence.

The execution trace shows var_A remaining constant at 100 while var_B increases: 1000 → 1001 → 1002 and so on. This demonstrates that the simulator correctly handles the swap operation, evaluates conditions properly, and executes operations as specified, even when the FSMD design itself does not converge efficiently.

Conclusion

The FSMD simulator is implementing core core execution semantics for FSMDs described in XML successfully. The three test cases verify basic functionality (Test 1), external stimuli handling (Test 2), and custom logic ith input manipulation (Test 3). Python's dynamic evaluation and dictionary structures enable a concise implementation while maintaining readability. The simulator provides a clear view cycle-by-cycle FSMD evolution and correctly executes all test cases, demonstrating it's suitability as an educational tool for understanding FSMD concepts.

Discussions & Limitations

The simulator is quite straight forward but comes with important design choices and limitations:

Limitations and Possible Improvements:

- ➔ Low to non-existent error handling: When errors occur, the simulator provides limited diagnostic information about which transition or operation caused the problem. Therefore, adding enhanced debugging features like breakpoints, single-step execution, and variable watch expressions.
- ➔ Flat namespaceing: All variables, inputs, operations and conditions share global dictionaries with no scoping or modularity support.