Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Asbjørn Steensland

# Universal Programmable State Machine

Master's thesis in Electronic Systems Design and Innovation
Supervisor: Per Gunnar Kjeldsberg
June 2019

NTNU
Norwegian University of Science and Technology

Asbjørn Steensland

# Universal Programmable State Machine

Master's thesis in Electronic Systems Design and Innovation
Supervisor: Per Gunnar Kjeldsberg
June 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

NTNU
Kunnskap for en bedre verden

# Assignment text

**Candidate name:** Asbjørn Steensland

**Assignment title:** Universal Programmable State Machine

**Assignment text:**

Currently, it is standard design practice to have individual custom made finite state machines (FSMs) that control use of each Mixed Mode IPs (like regulators, PLLs ...) in a design. The assingment will investigate the possibility of having an universal programmable FSM that can be used for all these, and possibly other IPs. Details:

- List advantages and disadvantages for an FSM vs a Microprocessor
- Investigate how such an FSM can be implemented
- Investigate how such an FSM can be programmed
- Investigate how such an FSM can be verified
- Investigate how such an FSM can be hooked up with the overall system
- Investigate how such an FSM can be configured
- Implement an example design based on the investigations outlined above

**Assignment proposer/Co-supervisors:** Anja Dekens and Conrad Foik, Nordic Semiconductor ASA

**Supervisor:** Per Gunnar Kjeldsberg, Department of Electronic Systems, NTNU

# Abstract

FSMs are used extensively in digital hardware designs. As the market for ICs and SoCs is increasing rapidly, the designs must follow. FSMs are often custom made and hardwired for each implementation. Thus, their behavior cannot be changed after manufacturing, and it takes time to design each FSM. This thesis investigates a Universal Programmable FSM (UPFSM), which can replace existing FSMs.

An FSM has a state set, a number of inputs and outputs, a state transition function, and an output function. The state transition function finds the next state based on the current state and the input signals. A Moore machine FSM calculates the output based on the current state, while a Mealy machine FSM also uses the input signals.

Two solutions for the UPFSM has been prototyped and investigated:

The LFSM is a LUT based solution which is similar to microprogrammed control. It stores the next state and the output in a LUT which is indexed using the current state and the inputs. The LUT is connected to a bus, allowing a microprocessor to write a new program to LFSM.

The SWFSM is a software-based solution. The input and output signals for the "FSM" are routed to registers accessible by a microprocessor. The microprocessor runs a program in software that read the input register, computes the next state and outputs, and write to the output register.

Both these solutions are used to replace an existing FSM in TD, an existing test-design. TD is part of a larger test-chip, TC, which has a microprocessor.

A comparison between the original hardwired FSM, the LFSM, and the SWFSM has been carried out. The original hardwired FSM is not programmable, but easy to use, has a small area, and provide an output every clock cycle. The LFSM is equally fast as the hardwired FSM but not easy to program and require more area. For the FSM in TD, the LFSM is 3.71 times (Moore) or 5.6 times (Mealy) larger than the original hardwired FSM in area. The LFSM area increases exponentially with FSM size. A "666" {#input bits, #state bits, #output bits} Moore LFSM is 441 times larger than a "222" Moore LFSM. The SWFSM require little area when the microprocessor is not considered, and it is easier to program. It is, however, very slow compared to the others, using 992 microprocessor clock cycles to read the input and write the output.

A UPFSM will always be a compromise between area, speed, ease of use, and complexity. However, the programmability can be very useful in certain designs, and this thesis has shown some possible ways to implement it.

# Sammendrag

Tilstandsmaskiner (FSM-er) blir utstrakt brukt i digital maskinvaredesign. Ettersom markedet for integrerte kretser og brikke-systemer er økende må designene følge etter. FSM-er blir ofte spesiallaget og hardkodet for hver enkelt implementasjon. Følgende kan ikke oppførselen endres etter at brikken er produsert. I tillegg tar det tid å designe hver FSM. Denne oppgaven undersøker en Universell Programmerbar FSM (UPFSM) som kan erstatte eksisterende FSM-er.

En FSM har et sett med tilstander, et antall inn- og utganger, en overgangs- og en utgangsfunksjon. Overgangsfunksjonen returnerer neste tilstand basert på nå-tilstanden og inngangsverdiene. Utgangsfunksjonen returnerer utgangsverdiene. For en Moore-maskin er denne kun avhengig av nå-tilstanden, mens en Mealy-maskin i tillegg er avhengig av inngangsverdiene.

To løsninger for en UPFSM er blitt prototypet og undersøkt:

LFSM-en er basert på en oppslagstabell (LUT), og minner om mikroprogrammert kontroll. Den holder neste-tilstanden og utgangsverdiene i en oppslagstabell som blir indeksert av nå-tilstanden og inngangsverdiene. LFSM-en er koblet til en buss, slik at en mikroprosessor kan skrive et nytt program til oppslagsen.

SWFSM-en er basert på programvare. Inn- og utgangssignalene til FSM-en er forlenget til registre som kan leses og skrives til av en mikroprosessor. Mikroprosessoren kjører et program som leser inngangsverdiene, regner ut neste tilstand og skriver utgangsverdiene til registerne.

Begge disse løsningene er brukt til å erstatte en eksisterende FSM i TD, et eksisterende test-design. TD er en del av en større test-brikke, TC, som har en mikroprosessor.

Den originale hardkodede FSM-en, LFSM-en og SWFSM-en er sammenlignet. Den originale FSM-en er ikke programmerbar, men er lett å bruke, krever lite areal og genererer utgangsverdier hver klokkesykel. LFSM-en er like rask, men er vanskelig å programmere og krever mye areal. For FSM-en i TD, bruker LFSM-en 3.71 ganger (Moore) eller 5.6 ganger (Mealy) mer areal enn den originale FSM-en. Arealet øker eksponensielt med størrelsen til FSM-en. En Moore LFSM med ”666” {#inngangssignaler, #tilstandsbits, #utgangssignaler} er 441 ganger større enn en ”222” Moore LFSM. SWFSM-en bruker mindre areal (sett bort i fra mikroprosessoren), er er enklere å programmere, men den er veldig treg siden den bruker 992 klokkesykler til å lese inngangen og skrive til utgangen.

En UPFSM vil alltid være et kompromiss mellom areal, ytelse, brukervennlighet og kompleksitet. Men, programmerbarhet kan i visse design være veldg nyttig, og denne oppgaven har vist noen mulige måte å implementere dette på.

# Preface

This master thesis is the final assignment for me as a student at Electronics Systems Design and Innovation an NTNU. Doing a thesis of this scope is not something I have done before, and it has been challenging and professionally rewarding. The experiences from a project assignment I did the previous semester has been useful for structuring the work and grasping the task, even though this is a different subject.

I want to thank all persons involved in helping me carry out this thesis; supervisor Per Gunnar Kjeldsberg at Department of Electronic Systems, NTNU, and supervisors Anja Dekens and Conrad Foik from Nordic Semiconductor. Also, thank you to Nordic Semiconductor engineers Knut Austbø, Vinodh Gunasekaren and Berend Dekens for taking your time to help me even though you had no official supervisor role. Nordic Semiconductor, as a company, also have my gratitude for providing working place, tools, and great support.

Trondheim 09.06.2019
Asbjørn Steensland

# Contents

# List of Figures

# List of Tables

# Abbreviations

**ALU** - Arithmetic Logic Unit

**CAR** - Control Adress Register

**CPU** - Central Processing Unit

**FSM** - Finite State Machine

**HDL** - Hardware Description Language

**IC** - Integrated circuit

**IoT** - Internet of Things

**IP** - Intellectual Property

**LFSM** - Lookup table Finite State Machine

**LUT** - Lookup table

**MMI** - Mixed Mode Intellectual Property

**RTL** - Register Transfer Level

**ROM** - Read Only Memory

**SWFSM** - Software Finite State Machine

**SoC** - System on Chip

**TC** - Test Chip

**TD** - Test Design

**UPFSM** - Universal Programmable State Machine

# 1 Introduction

## 1.1 Motivation

Our daily lives today are depending on digital electronics and integrated circuits (ICs). In the future, the demand for ICs will continue to increase. Internet of Things (IoT), where electronic devices are connected to the internet, will fuel this demand for the foreseeable future. In 2018, 7 billion IoT devices were connected, and this is predicted to increase to 22 billion devices in 2025 [1], [2]. Innovation and smart design are critical factors in order to fulfill the demands. Since Moores law is ending [3], smart design is even more critical. Moores law used to be a driving element in the industry, and designers must now find other means to develop the technology further. Another aspect is the increased complexity of the designs. In order to be competitive and meet time to market demands, it is beneficial to reduce the design time. This thesis aims at both provide innovations into the field of state machines and help reduce design time.

Digital logic can be classified as sequential or combinatorial. In a sequential system, the behavior is determined by the inputs to the system and the current state of the system. This indicates that the system uses the previous events to determine the current state and thus the behavior. Such systems cannot be described with a truth table and are instead described as a finite-state machine (FSM) [4].

In this thesis, the focus will be on different implementations of FSMs used to control other circuits or components of an IC. Usually, such FSMs are hardwired and custom made to each application and implementation. Designing a custom FSM each time it is needed is time-consuming and inflexible. The lack of flexibility arises from the fact that the workings and behavior of the FSM cannot be changed after manufacturing of circuit. Being able to change the FSM behavior can be useful if the circuit is not working as expected. The central concept in this thesis is thus to present a Universal Programmable FSM (UPFSM), which can be reprogrammed after manufacturing and can be used to replace hardwired FSM. The FSM does not have to be programmable in run-time and programmability at reset or power-up is sufficient.

Such a UPFSM can be implemented in software and use microprocessors to execute the software [5], [6]. This kind of implementation has the advantage of being flexible, relatively easy to use, and can do other tasks than just the FSM tasks. However, microprocessors are slow and costly in terms of area, complexity, and power consumption, so hardware FSMs are typically used. Instead of using a full microprocessor, it is possible to create something in between a microprocessor and a custom hardwired FSM, which can remedy a microprocessor's drawbacks of cost and speed.

## 1.2 Project description and limitations

This thesis describes the development and design of two possible solutions for a UPFSM. The two are a hardware-based solution and a software-based solution. These solutions are made to replace an existing FSM in a current IP-design, but they are designed with the ability to replace other FSMs as well. Furthermore, these solutions will be compared to each other and a hardwired FSM. The evaluation is based on area, throughput and speed, power consumption, and complexity and ease of use.

The hardware-based design, is implemented as an intellectual property core (IP) [7] for use in existing designs. It is not fully verified but is used to replace an existing hardwired FSM in an analog mixed-mode IP (MMI) which has a testbench for verification.

The software-based design, is based on forwarding the input and output signals to a microprocessor in a System-On-Chip (SoC). This microprocessor runs a software program that acts like an FSM. The software-based FSM is also used to replace the same hardwired FSM.

The assignment is proposed by semiconductor company Nordic Semiconductor, and existing designs and frameworks by Nordic semiconductor are used in this thesis.

## 1.3 Objectives and main contributions

The objectives in this thesis are:

- Compare state machines and microprocessors.
- Replace FSMs in an existing IP with a UPFSM.
- Evaluate and compare the different UPFSM solutions with the existing hardwired FSM implementation. The evaluation criteria are area, speed and throughput, power consumption, and ease of use.

The work and evaluation will contribute to an understanding of different ways to implement FSMs and use of them in digital circuits. This includes the compromises between area and speed, and how the compromises are affected by the FSM size. Besides, it may provide more ideas of FSM implementations and open up for other optimizations.

## 1.4 Report outline

The report is divided into the following chapters:

**Section 2: Theory** presenting relevant background theory on automata theory, FSMs,

microprocessors, microprogrammed control, and look-up-tables (LUT).

**Section 3: Design tools, previous work, and related work** describing the design tools and language used in this project and presents the previous work used for the UPFSM. In addition, an overview of work related to this project is presented.

**Section 4: UPFSM development and design** presenting the development and design of the hardware-based and software-based designs.

**Section 5: Testing, results, and discussion for the UPFSM** presenting the testing, results obtained, and a discussion for each of the two designs.

**Section 6: Evaluation and discussion** presenting an evaluation and comparison of the different ways to implement FSMs and suggesting topics for future work.

**Section 7: Conclusion** summarizing the findings in the thesis.

**References** list the references used in this thesis.

Additional information, such as source code are attached in the Appendices A to C.

# 2 Theory

This chapter presents and describes relevant background theory used in this report.

## 2.1 Automata theory

Automata Theory is a theoretical branch of computer science. In the 20th century, mathematicians began developing machines, both theoretically and physically, which completed calculations faster and more reliably than humans [8]. The word *automaton* means performing certain processes automatically and is closely related to the word "automation". An *automata* is a simple machine which can be used to perform the logic computations in such processes. Automata give computer scientists an understanding of how machines compute functions and solve problems. More importantly, they provide insight into what it means for a function to be *computable* or for a question to be *decidable* [8].

Abstract models of machines that perform computations on input signal changes by moving through a series of states or configurations are called automatons. A transition function determines the next configuration or state on the basis of a finite portion of the present configuration. This transition function is used in each state of the computation. The Turing machine is the most general and powerful automata [8].

The primary objective of automata theory is to develop methods which help to describe and analyze the dynamic behavior of discrete systems where signals are periodically sampled. The construction of the system in terms of storage and combinatorial elements determines the behavior. The characteristics typically include [8]:

- Inputs: sequences of input symbols from a finite set $I$ of input signals. $I = \{x_1, x_2, x_3...x_k\}$, where $k$ is the number of input signals.
- Outputs: sequences of symbols from a finite set $Z$ of output signals. $Z = \{y_1, y_2, y_3...y_m\}$, where $m$ is the number of output signals.
- States: a finite set $Q$, which is defined by the type of automaton

There are four major families of automatons:

- Finite-state machine (FSM)
- Pushdown automata
- Linear-bounded automata
- Turing machine

The above families can be seen as a hierarchy, where the simplest automaton is the FSM, and the most complex is the Turing machine [8]. Besides, it can be said that the Turing machine is an FSM, while the FSM is not a Turing machine. The Turing machine will be briefly described in Section 2.2.1.

## 2.2 Finite State Machines

Sometimes, the inputs to a system do not contain enough information to describe the system behavior. Modeling such systems require an internal state. If the number of states in a system is finite, it is a finite state system [9]. A finite state machine (FSM) is an automaton in which the state set Q contains a finite number of elements. FSMs are abstract machines, and consists of a set of states (Q), a set of input signals (I), a set of output signals (Z), a state transition function, and an output function. The state transition function returns the next state and a new set of output events by using the current state and an input event. In the most general form, the state transition function can be seen as a function which maps an ordered sequence of input signals (I) into a corresponding sequence of output signals (Z):

$$I \rightarrow Z \tag{1}$$

A finite state machine is a sequential system which provides an output based on the following five components, which can be described by the following definition [8]:

$$M = (Q, I, Z, \delta, W) \tag{2}$$

where
$Q$: Finite set of states.
$I$: Finite set of input signals.
$Z$: Finite set of output signals.
$\delta$: State transition function, $\delta : Q \times I \rightarrow Q$.
$W$: Output function, $W : Q \times I \rightarrow Z$.

As mentioned, an FSM contains a finite number of states. Each state accepts a finite number of inputs, and each state has a state transition mapping function that describes what actions to perform for a change in the input signals. Thus, a change in the input signals may cause the machine to change states and change the output signals.

In order to explain some more concepts used in FSMs, consider an example of an FSM. It should have two states, S1 and S2, and one input signal (I) and one output signal (0).

If I=0, the FSM should go to state S0, and if I=1, the FSM should go to S1. In S0, O=0 and in S1, O=1. Since the outputs only depend on the current state, it would be a Moore machine. Moore machines and the counterpart, Mealy machines, is elaborated further in Section 2.2.2. This FSM could, for example, describe a door, where S0 is an open door, and S1 is a closed door. The input could resemble the door handle, and the output is the actual door position; closed or open. This can be described using a *state transition table*, which is presented in Table 2.1.

**Table 2.1:** State transition table for a simple FSM

| Input | Current State | Output | Next State |
|---|---|---|---|
| 0 | S0 | 0 | S0 |
| 1 | S0 | 0 | S1 |
| 0 | S1 | 1 | S0 |
| 1 | S1 | 1 | S1 |

The state transition table contains all necessary information to describe an FSM. The state transition table shows all the states, the transitions between the states, and what the output is in each state. Such a table is still not a formal description, but it is intuitive and easy to grasp. A state diagram illustrates the state transition table. The state diagram does not formally describe the FSM, but contains the same information as a formal description [10]. Figure 2.1 shows the state diagram for the example FSM.

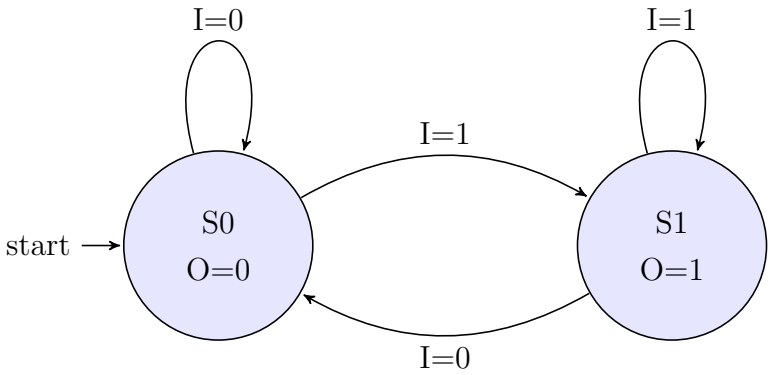



**Figure 2.1:** Simple state diagram

Using formal definition from Equation (2), the example FSM will be described as

$$M_1 = (\{S0, S1\}, \{I\}, \{O\}, \delta, \{W\}) \tag{3}$$

In Equation (3), the example state machine is called $M_1$, and it has the state set $Q = \{S0, S1\}$, the input signal $I = \{I\}$ and the output signal $Z = \{O\}$. The state transition function $\delta$ is described using Table 2.2, and the output function is described using Table 2.3.

**Table 2.2:** State transition function $\delta$. The next state is based on the current state and the input signal.

| | 0 | 1 |
|---|---|---|
| S0 | S0 | S1 |
| S1 | S0 | S1 |

**Table 2.3:** Output function $W$. The output signal is determined by the current state only.

| | 0 | 1 |
|---|---|---|
| S0 | 0 | 0 |
| S1 | 1 | 1 |

### 2.2.1 FSM versus Turing machine

As previously described, the FSM is the simplest automaton, e.g., the simplest way to describe a computational system. It is, therefore, not an adequate computation model for all types of systems. A Turing machine, on the other hand, is a lot more powerful.

A theoretical computer has two states for each bit, but within the computer, there is no theoretical limit of the number of components the computer interacts with. This means an FSM will not be sufficient to model a computer. However, more powerful automata, such as a Turing machine can be capable enough.

A Turing machine is an abstract machine essentially consisting of a "control unit", a "tape" and a "read-write device" [11]. The control unit, which essentially is an FSM, contains a set of instructions and is at any time in a certain state. The state set is finite. The tape is divided into squares, and each square can contain a symbol. The read-write device can operate on exactly one square of the tape. It can read an existing symbol on the tape, or overwrite what is on the square. The read-write device can move left or right on the tape. The tape can be infinitely long, giving the Turing machine infinite memory. [8]. Formally, the Turing machine is defined as a 7-tuple [12]:

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F) \tag{4}$$

where
$Q$: The finite set of states of the control unit.
$\Sigma$: The finite set of input symbols.
$\Gamma$: The complete set of tape symbols, which always is a subset of $\Sigma$.
$\delta$: The transition function $\delta(q, X)$, with the arguments state $q$ and tape symbol $X$.
$q_0$: The start state, which is a member of $Q$.
$B$: The blank symbol. This is in $\Gamma$, but not in $\Sigma$.

$F$: The set of accepting states, a subset of $Q$.

The Turing machine differs from an FSM in the fact that the Turing machine is capable of changing symbols on its infinitely long tape. The FSM's only memory is its current state, while the Turing machine has memory on the tape, which can be infinitely long. The reading and writing of the tape simulates the execution and storage used in computers and means the Turing machine can model all computations that can be calculated in modern computers [12].

### 2.2.2 Mealy and Moore FSM

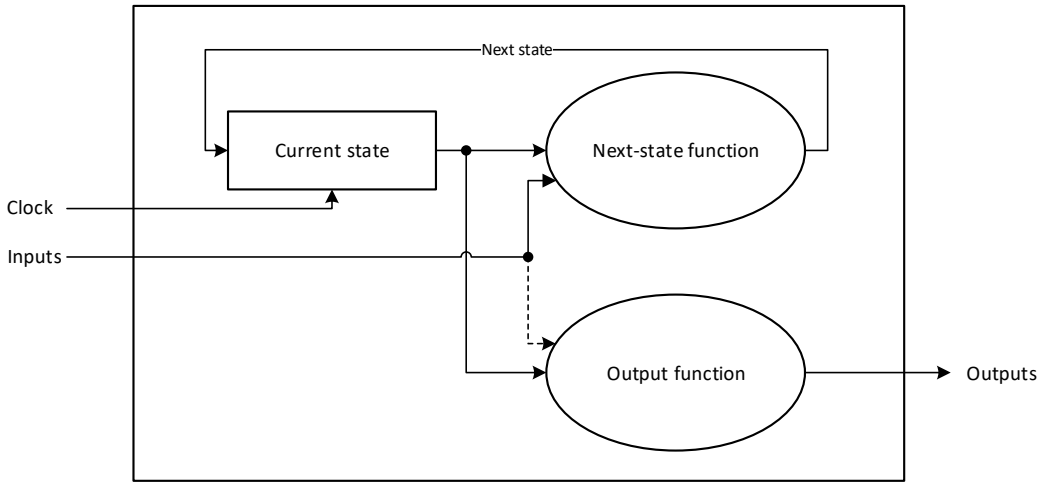



**Figure 2.2:** Model of the functions in a FSM. The dashed line only applies to Mealy machines.

The output from an FSM can be determined by the current state only, or a combination of the current state and the inputs. The first one is called a Moore machine, named after Edward Moore, who presented the concepts in a paper [13] in 1956. The latter machine is called a Mealy machine and is named after George H. Mealy [14]. In a Mealy machine, the outputs are determined by both the inputs and the current state. It follows that the output function uses both the current state and inputs as parameters. Figure 2.2 depicts the functions and components of an FSM and the difference between Mealy and Moore. The output function only depends on the current state for a Moore machine, while it also depends on the inputs for a Mealy machine.

### 2.2.3 Implementation of FSM in hardware

In a digital circuit, a state register holds the current state, and two logic functions compute the output and the next state which is to be fed to the state register. In a hardware-description language (HDL) such as SystemVerilog, the example FSM could be written

using enumerated types for the states [15]. A way to create the example FSM in SystemVerilog is presented in Listing 1. The SystemVerilog language is further elaborated in Section 3.1.1.

```
module example_FSM
(
    input logic clk,
    input logic rst,
    input logic I,
    output logic O
);
    typedef enum logic {S0, S1} stateType;

    stateType state, nextState;

    always_ff @(posedge clk or posedge rst) begin
        if (rst) state <= S0;
        else       state <= nextState;
    end

    always_comb begin
        case (state)
            S0: begin
                O = '0;
                if (!I) nextState = S0;
                else    nextState = S1;
            end

            S1: begin
                O = '1;
                if (I)  nextState = S1;
                else    nextState = S0;
            end
        endcase
    end
endmodule
```

**Listing 1:** HDL code for a simple FSM

In this code, one can see that the nextState function and output functions are combinatorial logic and is declared inside an *always_comb* block. The *always_ff* block contain the state register. A register is sequential logic and needs a clock for synchronization. This code will result in a circuit almost similar to the Moore machine version in Figure 2.2, but with some generalizations, as can be seen in Figure 2.3.

There are many more ways to create and optimize hardware implementations of FSMs. Many computer-aided design (CAD) programs can use either a graphical or textual representation of an FSM and produce an optimized implementation automatically [4].

## 2.3 Microprocessors

As presented in Section 2.1, it is possible to argue that a microprocessor can be modeled as an FSM automaton. In the real world, however, microprocessors are more general and capable than hardware FSMs. A microprocessor is an IC containing a central processing unit (CPU) [16]. The microprocessor is a register-based, clock driven digital IC

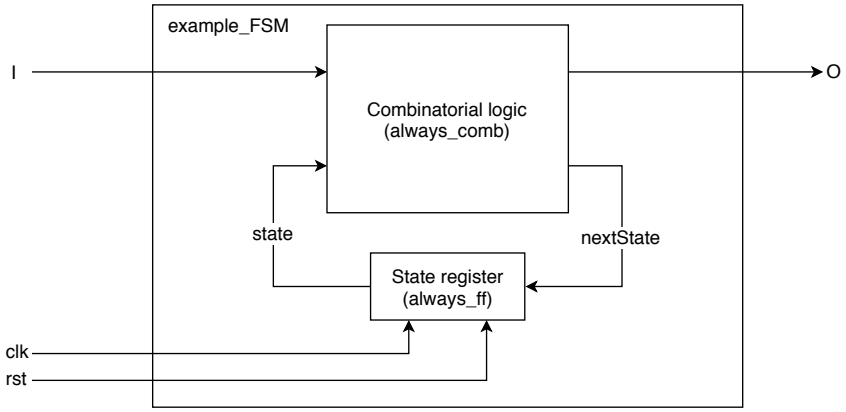



**Figure 2.3:** HDL implementation of the example FSM.

that accepts binary input data. The input data is processed according to instructions found in the memory and provides results as outputs. Depending on the instructions, the processing can be performed differently.

The logic circuitry of a microprocessor can be divided into two parts: the datapath and the control unit [17]. The datapath performs the actual executions of the data, such as adding numbers or writing to registers. The datapath thus contains the pipeline with its functional units, such as the arithmetic logic unit (ALU) and registers for the temporary storage of data. The control unit is responsible for controlling the datapath by setting control signals to the functional units and the peripheral units such as memory.

## 2.4 Microprogrammed Control

A control unit with its binary control values stored as words in memory is called microprogrammed control [18]. Each word contains a microinstruction that specifies one or more microoperations for the system. A sequence of such microinstructions is called a microprogram. Usually, this microprogram is fixed at design time and stored in ROM. However, it can be stored in RAM and loaded at system startup from non-volatile storage. A RAM solution is called writable control memory and would require read/write functionality [18].

Figure 2.4 show the general configuration of microprogrammed control. The "control address register" (CAR) specifies the address of the microinstruction in the ROM. One part of the microinstruction word is used to determine the address of the next microinstruction. The rest of the microinstruction is used for various control signals. The next-address information may be combined with various input signals in the "next-address generator", which generates the next address. The CAR and the next-address generator is often called a sequencer.

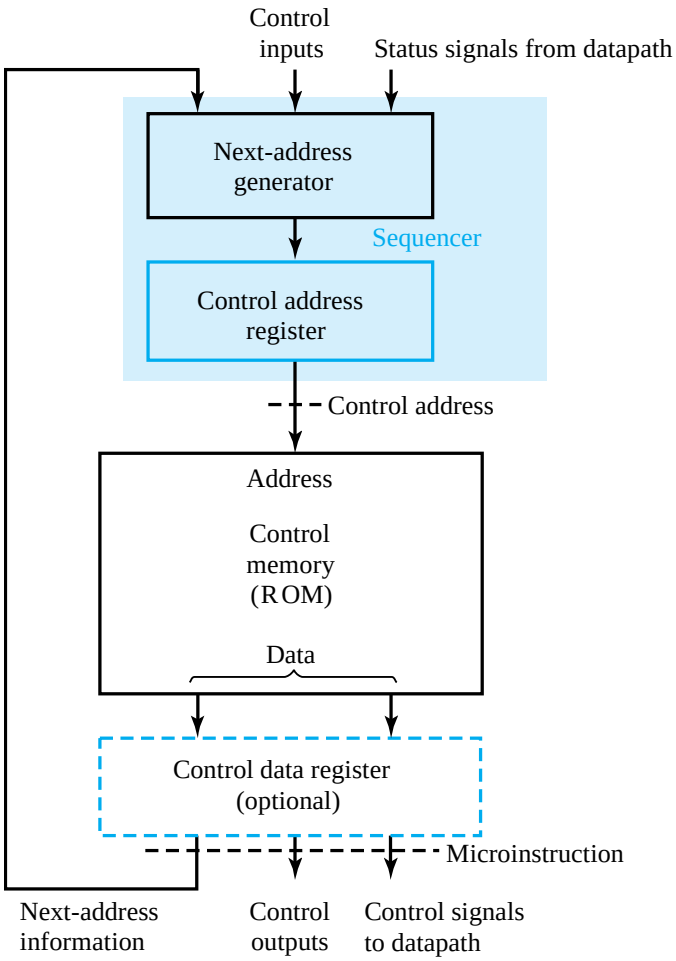



**Figure 2.4:** Microprogrammed control unit organization [18]

## 2.5 Lookup table

A lookup table (LUT) is a group of memory cells which, given a set of input values, contain all possible results of a given function [19]. The values of the function must be stored so that the output values correspond to the input values. A LUT with $n$ inputs will have $2^n$ single bit memory cells. LUTs are often used to encode logic functions in field-programmable gate arrays (FPGAs).

# 3 Design tools, previous work and related work

## 3.1 Design tools

This chapter presents an overview and a brief description of the design language and tools used in the work for this thesis.

### 3.1.1 SystemVerilog

SystemVerilog is a hardware-description language (HDL) [20]. It is used to model, simulate, and verify the functionality of digital circuits at levels of abstraction ranging from system level down to gate and switch level, including RTL. The following paragraphs present some important constructs and features used in this thesis.

**Module** is a basic unit of hierarchy in SystemVerilog. Modules contain declarations and functional descriptions and represent hardware components. They can be nested, meaning a module can be instantiated inside another module.

**SystemVerilog Interface (SVI)** is used to encapsulate communication between blocks, for example, between modules. This makes it useful for design reuse, increases code readability, and ease the maintenance of IPs [20]. In its purest form, an SVI is a bundle of nets and variables but can contain tasks and functions to model bus functionality.

**Assertions** are used for functional verification. Two types exist, immediate and concurrent assertions. Immediate assertions are used to test the value of an expression. Concurrent assertions are used to test properties. Properties are built from sequences, which describe behavior over time, and thus with respect to a clock. If an assertion is violated, an error is generated. Assertions are typically checked dynamically during simulations [20].

***always_comb* and *always_ff*** are the two main SystemVerilog constructs used in this thesis. They are used for combinatorial and sequential logic, respectively. [20]

***logic*, *enum* and *parameter*** are the most prevalent types used. *logic* is the same as *reg*, and has a user-defined size. *logic* is preferred in SystemVerilog [20]. *Enum* is used to compact the code and make it more readable. *Typedef* is used in combination with *enum* to define user-defined types. *Parameter* is used to giving names to constant values.

This can be used to define bus widths etc., which makes simulating different sizes and modifying the design easier.

### 3.1.2 Mentor Graphics QuestaSim®

QuestaSim® is a simulation tool provided by Mentor Graphics. It is used to simulate RTL designs and netlists, running testbenches, and help in verification of designs [21]. The waveform generator is used extensively in the design work in this thesis. The waveform can be used to measure the time between signals are set and see how the signals behave. It supports SystemVerilog assertions and coverage for verification. For full-chip simulations, version 10.7c_3 was used.

### 3.1.3 Industry standard synthesis tool

For synthesis of the different designs in order to get results for area, power consumption and throughput, an industry standard synthesis tool is used.

## 3.2 Previous work at Nordic Semiconductor

Both following sections describe work done by Nordic Semiconductor. This work is used as a foundation for development, design, and testing the different UPFSM implementations.

### 3.2.1 Test Design (TD)

TD is an IP from Nordic Semiconductor written in SystemVerilog. It is a mixed-mode IP (MMI) used to control an analog power regulator using digital logic. It is a part of a larger system for power supply to an SoC called Test Chip (TC). In this thesis, an FSM inside TD is used for experimentation and evaluation purposes.

TD is connected to the main bus structure on the SoC and can thus be enabled or disabled from software. If TD is enabled and a USB cable is connected, TD will initiate a startup sequence using the aforementioned FSM for the analog logic and notify the microprocessor when the USB power supply is ready and stable. The USB cable is in this context only regarded as a power source.

The FSM in question has 4 input signals, 3 output signals, and 5 states. The state transition diagram is seen in Figure 3.1. Only the input signals indicated at the transition arrows are used for that transition. All other input signals are don't care bits and not used.

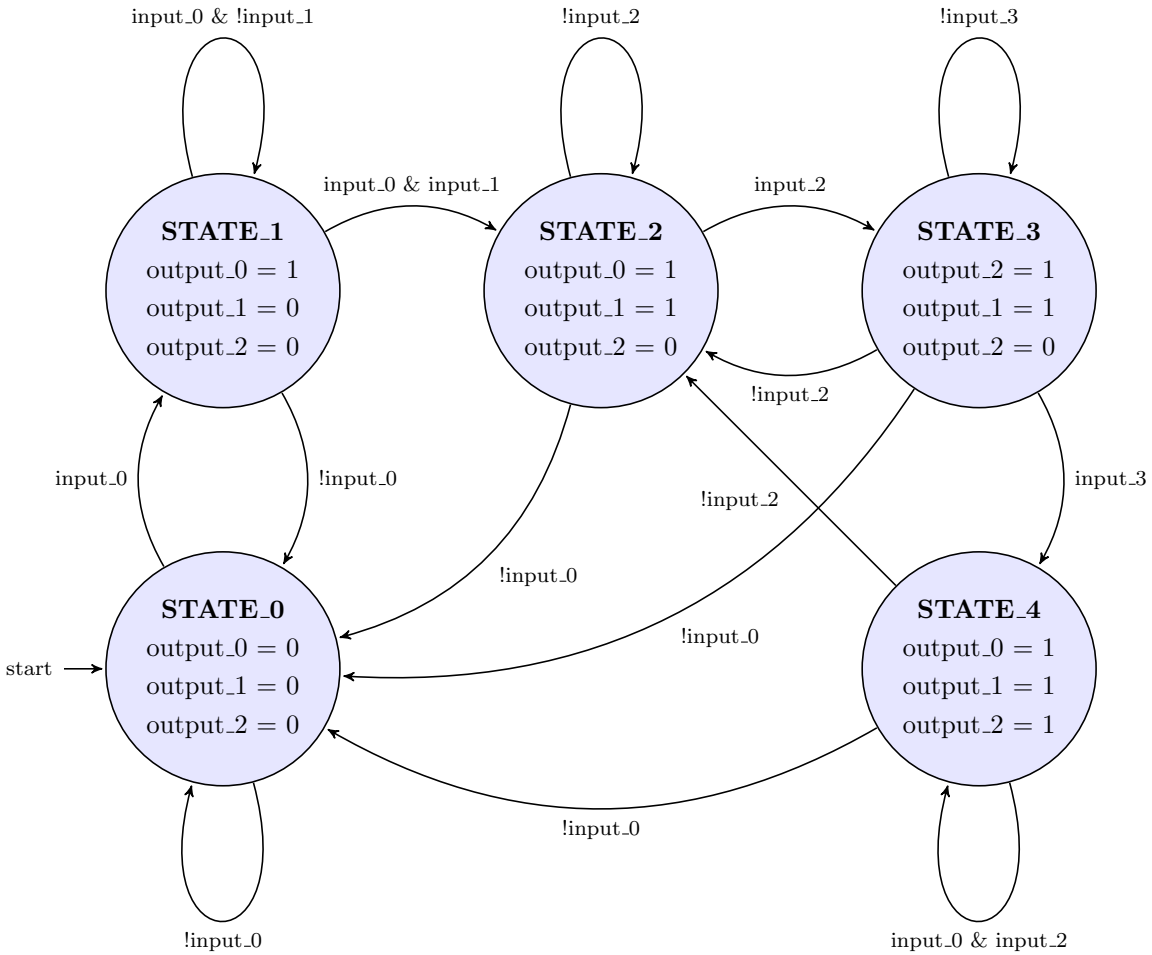



**Figure 3.1:** State diagram of TD.

The exact mechanisms controlled by this FSM are not crucial for this thesis. However, TD and its FSM was relevant for experimentation for programmable FSMs, as the physical, manufactured analog circuits may behave differently than the models used in the design work. Being able to change the FSM behavior would allow for optimizing the circuit for the analog circuits. The original FSM in TD is created similarly to the method presented in Section 2.2.3.

### 3.2.2 Test Chip (TC)

TC is an SoC from Nordic Semiconductor, in which TD is included. TC is a wireless communications SoC, with several peripherals and microprocessors. It is designed for low power operation. The microprocessor used in this thesis is from ARM®. The microprocessor is connected to a bus structure, which enables the microprocessor to access TD and other components of TC. The physical TC was not used; only a model was used to simulate the behavior of TC. The model of TC has an existing framework and toolchains

for running simulations and programs.

## 3.3 Related work

This chapter presents work which is related to this thesis. Some of this work is directly relevant for this thesis, as it has been the basis of the experimentation and evaluations. Some of the work is not directly relevant for this thesis, but it can suggest other ways of implementing FSMs and invoke other possible solutions.

### 3.3.1 Software based FSMs on microprocessors

Work in the field of software-based FSMs on microprocessors is relevant for programmable FSMs. In this thesis, the work done by the microprocessor architecture company ARM [5] and by Comer [6] is used. ARMs white paper is the most recent and relevant for this thesis, but the work of Comer is, albeit old, relevant as it presents parameters and what to consider when comparing conventional state machines to microprocessor-based state machines.

**ARM** proposed in a white paper that the microprocessor architectures developed by ARM can be used to replace hardware FSMs [5]. The paper presents several reasons to use software-based FSMs in SoC designs and how they can be implemented. The advantages of using software-based FSMs are:

- Increased flexibility: As the software-based FSMs reduce the risk of the project by allowing the FSM to be changed at any time in the project cycle. A hardware FSM cannot easily, if at all, be changed after production. Figure 3.2 shows how this flexibility compares to the other methods of designing FSMs.
- Debug methodology: Since the software-based FSMs use a program, conventional debug tools can be used to, for example, set breakpoints and halt the processor. This helps in optimizing the design.
- Combine software control capability to FSM design: Sequential operations are easier to handle, and functions/subroutine calls can be used to implement repetitive tasks. Also, assembly code can be used to optimize parts of the FSM operations, and the rest can be programmed in C.

The white paper then presents some design examples and what to consider for implementing an FSM on the different ARM Cortex-M architectures. It is challenging to meet the speed and time-critical requirements, and the paper also mentions workarounds and

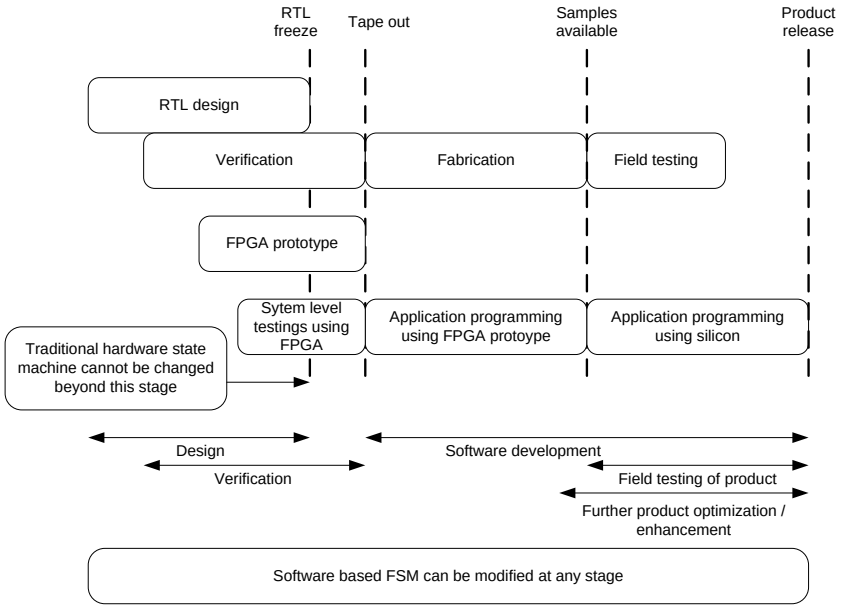



**Figure 3.2:** Timeline showing the flexibility of a software based FSM during the design stages [5].

propose methods to help with such challenges. However, many of these methods require high clock frequencies.

**Comer** considers the use of a microprocessor as a controller for digital systems [6]. The paper, published in 1987, suggests that a microprocessor can be considered as an FSM with each state being defined in terms of possible machine instructions. The approach is based on the following premises:

1. The state machine approach is used for the microprocessor controller. This means that the functionality of the controller is partitioned into states, which are implemented in software running on the microprocessor.

2. Since the same approach is used for both a controller running in a microprocessor and a controller in a hardware state machine, it allows for partitioning the functionality into software and hardware.

3. After all refinement has been done, a comparison between a conventional state machine and a microprocessor state machine can be carried out. A choice between them can thus be based on criteria such as cost, area, or performance.

Some key differences between them are presented. A conventional state machine computes the next state directly when the input changes, while a microprocessor must add the inputs to an accumulator which is used to branch to the next state. The addition to the accumulator involves at least one data transfer. Another difference is the timing,

where a conventional state machine remains in a given state for one clock cycle, and then moves to a new state or remains in the same. In a microprocessor, some states may only be for timing purposes, and others consider the inputs. A third difference is in the data manipulation or processing capabilities of the microprocessor, which opens up possibilities for storing data and use it for branching later and so on. A conventional state machine will require additional circuitry to do this.

The strengths of microprocessors are versatility, the possibility of more states without increasing system complexity, and the fact that they are programmable. The main weaknesses are speed and cost. The cost includes both the hardware and also the fact that microprocessors require programming, which in 1987 involved paying a manufacturer to program the ROM or buying an expensive ROM burner.

The concluding remarks are that a microprocessor controller is limited to lower frequency operations and are more expensive for smaller systems. For large systems with many states or that require flexibility in terms of changing the sequence of control signals, using a microprocessor controller might be reasonable.

### 3.3.2 Programmable FSMs using custom processor architectures

Using a custom processor architecture to implement a programmable state machine has been done in the past. Wangyang et al. [22] and Hatta et al. [23] has done this, and their work is briefly presented below.

**Wangyang et al.** created a programmable state machine for packet processing [22]. It is called a PSM and is intended to bridge the gap between a hardware FSM and a full processor. The PSM is a 4 stage pipeline with 18 instructions which are classified as register type, immediate type, and branch types. The application example is for packet processing. The PSM replace an FSM which is used for parsing packet headers. The main drawback with the PSM instead of a hardware FSM is speed, but it is less complicated and costly than a full processor and retains the advantages of programmability.

**Hatta et al.** also created a programmable FSM which they call the P-FSM [23]. It is designed to minimize both the logic and memory area using a specific architecture for state management of communication protocols. The P-FSM is a state processor capable of handling various types of state diagrams. The instruction memory can be changed and used to implement additional state management. A sequencer is used to index the instruction memory. The P-FSM is used as a hardware accelerator on a communications SoC, where it placed between the embedded processor and *dynamically reconfigurable*

*hardware for protocol processing* (DRHPP). It must be programmable in order to handle various communication protocols.

The P-FSM contains a sequencer used to address the instruction memory. The sequencer is also used as a step counter to control activation of an instruction processor. The instruction memory contains several FSM programs, and the sequencer thus indexes the correct FSM instructions. The instruction processor consists of an instruction decoder, calculation block, forwarding block, register-comparison block, event-comparison block, and two MUXes.

According to the article, it is, for a network communications application, 90 times faster than an ARM processor and from RTL synthesis only 1.5% of the area of a conventional communications SoC.

# 4 UPFSM development and design

The goal of this thesis is to make a Universal Programmable State Machine (UPFSM) which can replace FSMs in existing IP designs and is possible to reprogram after the SoC has been manufactured. The thesis will also include an evaluation of possible ways to implement a UPFSM and compare it to a hardwired FSM, in terms of area, speed, power consumption, and ease of use.

An existing IP, TD, and SoC, TC, is used to develop and test the UPFSM. TD and TC are presented in Section 3.3. TD contains a hardwired FSM from before which is used to control an analog MMI (mixed mode IP). This analog MMI is a power regulator which gets its power supply from a USB cable. The existing FSM is therefore called the "USB-FSM". A simplified overview of TD and the USB-FSM can be seen in Figure 4.1. A realistic aim in making the UPFSM is to be able to replace the USB-FSM and to be able to program it. The USB-FSM is only an example of an FSM which can be replaced, meaning the UPFSM should be able to replace most other FSMs as well.

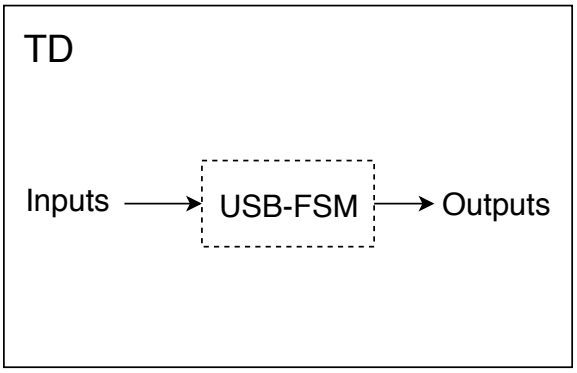



**Figure 4.1:** The existing TD with the hardwired USB-FSM. The dashed lines indicates that the USB-FSM is not an IP, but is hardwired in TD.

A UPFSM can be implemented in many ways. Two possible starting points are using a hardware-based design or a software-based design. A hardware-based design involves developing a system that is programmable, flexible, and can replace existing design. A software-based design means implementing the UPFSM in software that is running on a microprocessor. The microprocessor can be hooked up to the existing FSM connections. Investigating both of these solution spaces would allow for interesting evaluation and comparison. Therefore, two prototype designs for replacing the USB-FSM are implemented. The development of the two methods are presented in Section 4.1 and Section 4.2.

## 4.1 Hardware-based UPFSM

A hardware-based design can be inserted in the place of the original FSM. In the case of TD, this will involve inserting a system to replace the original hardwired USB-FSM with a UPFSM. To be programmable, the UPFSM must most likely have a bus interface which is used for programming. The idea is illustrated in Figure 4.2.

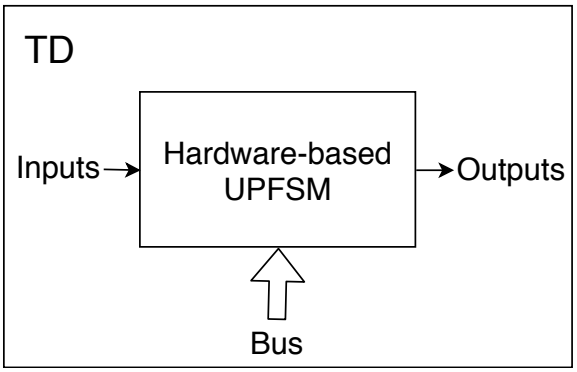



**Figure 4.2:** TD with the hardware-based UPFSM.

The inputs and outputs from the original FSM are connected to the hardware system. It would, therefore, be useful for the system to be parameterized in such a way that it can be used to replace different sizes of FSMs. Section 4.1.1 describes how the hardware-based system was investigated and developed, and Section 4.1.2 describes the system that was implemented based on the findings in the investigation.

### 4.1.1 Development

For learning and realizing what goes into an FSM in general, Python was used to model the USB-FSM. The goal for this modeling was to determine what functionality should be made programmable and to aid in making the UPFSM in hardware. This includes how to read the inputs and write to the outputs, as well as how to implement the next-state and output functions. However, it proved to be challenging to use Python for this task. One problem is that Python does not contain switch-statements, which are useful for describing the FSM. Instead, a bundle of "if-else" statements had to be used, and this made the code more confusing and did not help much in terms of making the USB-FSM more understandable. A lower level language such as C would probably have been a better choice since switch-statements can be used and the language provides more control to what data types are used.

As the Python modeling was not as fruitful as one could have hoped, prototyping in SystemVerilog was done instead. A common approach, as mentioned in Section 2.2.3, was used in the beginning. This approach utilizes an enumerated variable for the different

states and a switch-case statement with some if-else statements to compute the next state and outputs. Firstly, a simple FSM was made, similar to the example FSM in Section 2.2.3. This was done in order to get a better understanding of how this way to write an FSM worked in SystemVerilog and how it can be made programmable. More research was, however, needed to find ways to make a programmable system. Two main possibilities were discovered; a custom instruction set processor or a microprogrammed control system.

A processor is programmable by nature, as described in Section 2.3. As mentioned in Section 2.2.1, a microprocessor can be modeled as a Turing machine, which is a more powerful automaton than the FSM. Therefore, having a custom instruction set processor architecture was considered, as it would be programmable, and flexible. Instead of a general microprocessor, a processor reduced to the components and instruction set needed to run a programmable FSM could be used. Such a system had been implemented by Wangyang et al. [22] and Hatta et al. [23]. Their solutions are presented in Section 3.3. However, designing a custom instruction set architecture is complicated, and due to limited time for this thesis, it was decided to find other solutions to make a hardware-based UPFSM.

Microprogrammed control was the other possibility that was discovered. Since the USB-FSM is used to control another system, this inspired to research other ways to implement control circuits. In [4], it was discovered that hardwired FSMs can be used to control microprocessors. [4] also presented microprogrammed control as a an alternative way to control microprocessors. Microprogrammed control is presented in Section 2.4.

Microprogrammed control works on the principle of a memory with control words stored as microinstructions. This memory is typically a ROM. A sequencer finds the address of the next microinstruction based on information from the following microinstruction and some input signals to the control unit. Microprogrammed control is, in fact, an FSM, with each microinstruction containing the output values and the next state information. However, it was considered not to be particularly universal, as an interface with the sequencer is needed, and the next state is not only generated by the ROM but also linked to the program running in the processor itself. Also, the ROM is by definition, not writable, so it is not possible to reprogram or reconfigure the ROM. As mentioned in Section 2.4, RAM can also be used in writable microprogrammed control. Using this would make it more programmable and flexible. However, implementing a RAM with functionality for reading and writing seemed slow and unnecessarily complicated. Using registers instead would make likely make for a faster and simpler design.

Using registers to store functionality was further researched. The motivation was to use general ideas from microprogrammed control and simplify it. Therefore, the concept of a LUT was discovered. LUT is presented in Section 2.5. A LUT is a way to map an input to

an output. This would be beneficial since it avoids the addressing and sequencing needed for a microprogrammed control solution with RAM. A LUT-based design would also be less complicated than a custom processor architecture. Therefore, it was decided to move on with designing a LUT-based UPFSM called the LFSM, as in "LUT-FSM".

### 4.1.2 LFSM design

As mentioned, the LFSM is a variation of writable microprogrammed control. Figure 4.3 show an overview of the LFSM. As the figure depicts, it consists of three main parts; a LUT, write logic, and a state register. Each part and how the LFSM is programmed is described in the following paragraphs.

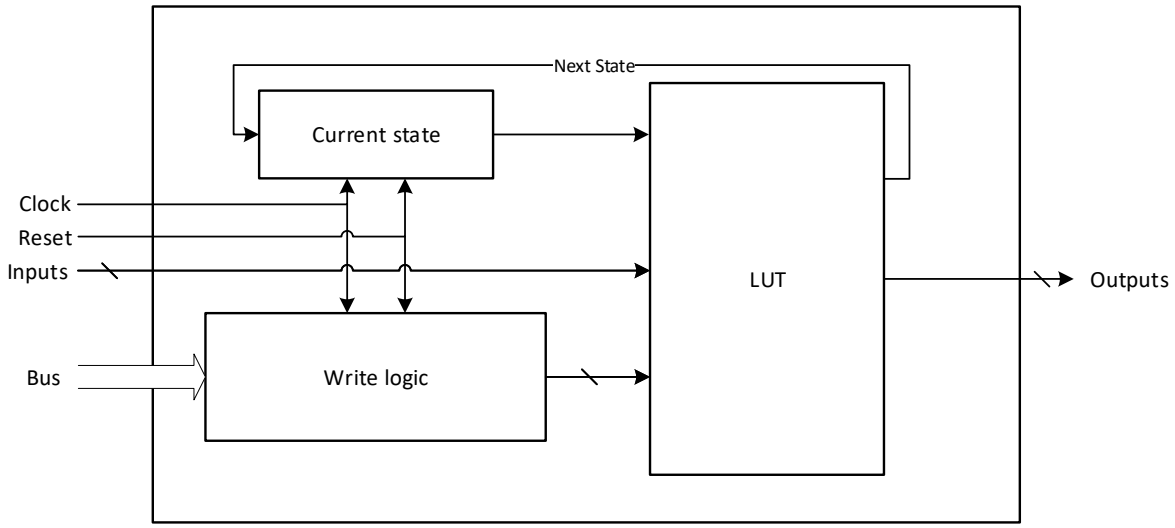



**Figure 4.3:** Overview of the LFSM and its three main parts.

**LUT** From Section 2.2, it is clear that an FSM has a transition function which returns the next state based on the current state and the input. In addition, the output is returned by an output function, which, depending on if it is a Moore or Mealy machine, is based on the current state and the inputs. A state transition table can describe this dependency. In the LFSM, the state transition table is implemented in a three-dimensional LUT. The LUT thus replace the state transition and output functions.

Figure 4.4 illustrates the layout of the LUT. At each position in the LUT, there is a word containing the next state bits and the output value. Indexing the LUT with the combination of the value of the current state bits and the value of the input signals, the position returns the corresponding output value and the next state. The LFSM with a LUT resembles microprogrammed control, except instead of using an address generator, the LUT uses current state and the inputs directly. The LUT size is decided by the number of input signals, output signals, and states needed for the desired FSM.

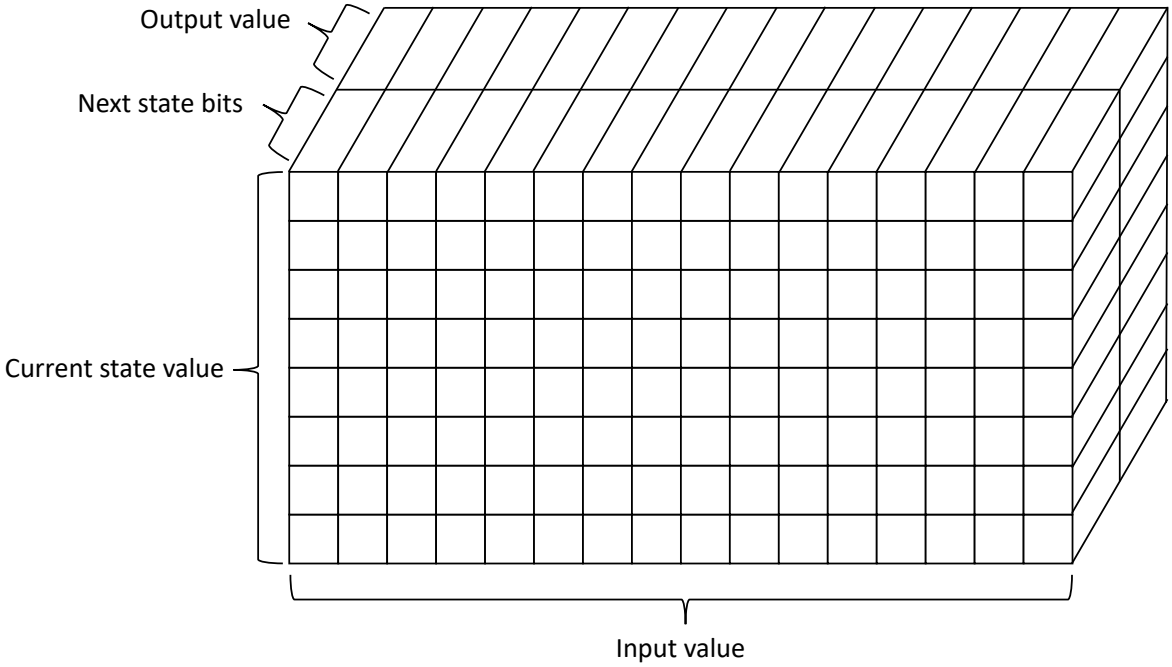



**Figure 4.4:** The layout of the LUT. The size in this figure correspond to the USB-FSM.

```
logic [2**MAX_NUM_STATE_BITS-1:0][2**MAX_INPUT_WITDH-1:0]
  ↪  [MAX_OUTPUT_WITDH+MAX_NUM_STATE_BITS-1:0] state_table;
```

**Listing 2:** Declaration of LUT.

Listing 2 show how the LUT named "state_table" is declared in SystemVerilog. As one can see, there are three parameters to specify the size of the LUT. Equation (5) shows how the size of the LUT is determined by the number of inputs signals, output signals, and state bits.

$$Size = 2^{(I+Q)} * (Z + Q) \tag{5}$$

where:

- $Q$ = **MAX_NUM_STATE_BITS** which specifies the number of bits needed to represent the state set.
- $I$ = **MAX_INPUT_WITDH** which specifies the number of input bits.
- $Z$ = **MAX_OUTPUT_WIDTH** which specifies the number of output bits.

The LUT must be sized to this in order to accommodate all possible combinations of input signal values and state bit values,and to store a word with next state bits and output signals for each position. Figure 4.4 corresponds to the USB-FSM, which has Q=3, I=4, and Z=3. This gives 16 possible input signal values and 8 possible state

values. Each combination of an input value and a state value holds a 6 bit word (Q+Z). The USB-FSM would, therefore, need a LUT size of 768 bits.

Since the LUT is generated in hardware, it must be sized large enough to accommodate all possible FSM sizes that might be needed for the UPFSM in the future. It is not possible to resize the LUT after the design is manufactured. However, it is possible to use fewer states, input signals, and output signals than specified by the parameters. This would, of course, lead to an area penalty, which can be very high for large FSMs with many states, inputs, and outputs. Besides, the LUT size is determined by the number of bits needed to represent the state set. For the USB-FSM, which has 5 states, Q=3 bits is must still be used to represent the state set in the LUT. 3 bits is enough for 8 states, and this means the LUT is excessively large.

**Write logic** The LUT is writeable and can be programmed. The programming is done by connecting a bus interface to the LFSM and use logic to write to the LUT. Using this bus interface, an FSM configuration can be loaded into the LUT, which obviously will change the contents in the LUT. The new content in the LUT results in a different state transition and output functions, and the FSM will react to the inputs differently. The bus is connected to the main bus structure of the chip, and the on-chip microprocessor can load the FSM configuration from non-volatile memory by software at startup.

The bus interface has both read and write capabilities, but only the write functionality is used in the LFSM. The bus has an addressing scheme which is used to write and read from other IPs. The bus data width is 32 bits, which for this design of the LFSM means each word in the LUT only can be 32 bits wide. Due to this, the maximum number of output bits and state bits combined cannot exceed 32 bits. However, the word length issue can most likely be solved by, for example, using a two-stage write process which loads each position with two bus transfers. This was not implemented for in the LFSM due to limited time and since it is not required for the USB-FSM.

Listing 3 show the SystemVerilog code for how the LFSM configures the LUT from the bus. Since it must be synchronized with the bus, the write logic is in a sequential *always_ff* block. The writing must be done by a bus master, such as a microprocessor, which uses a software program to write the LUT at chip startup or reset. Due to limited time in the work for this thesis, a bus driver in a testbench has been used for writing to the LFSM. Writing starts by writing 32'hFFFFFFFF to address 32'hFFC in the LFSM. This toggles a register which tells the LFSM that it is in programming mode. While it is in programming mode, it will not function as an FSM, and all the outputs will be zero. Then, at each clock tick, a new word is written to the LUT from the bus, and the counters increment and writing to the next position is performed. This iteration continues until the entire FSM configuration has been written. After the FSM configuration writing finishes,

```
// Used for iterating through LUT
logic [MAX_NUM_STATES-1:0] i;
logic [MAX_NUM_INPUTS-1:0] j;

// Write LUT
always_ff @(posedge clk or posedge arst) begin
  // LUT set to zero on reset, and index variables are set to zero
  if (arst) begin
      state_table <= '0;
      i <= '0;
      j <= '0;
  end
  else begin
    case(busAddr)
      32'hFFC : begin //Set FSM in programming mode by sending FFFFFFFF to address FFC
        if(busWe && busDo == 32'hFFFFFFFF) begin
          if (programming_mode) programming_mode <= '0; //Toggle programming mode
          else programming_mode <= '1;
        end
      end
      //For every entry in state table to be indexable using inputs and state bits, the below musbe done:
      LFSM_BASE : begin
        if(busWe) begin
          if(programming_mode) state_table[i][j] <= busDo[MAX_NUM_OUTPUTS + MAX_NUM_STATES - 1 : 0];
          // Incrementing indexes
          j <= j + 1;
          if(j >= 2**MAX_NUM_INPUTS - 1) begin
            i <= i + 1;
            j <= '0;
            if(i >= 2**MAX_NUM_STATES - 1) begin
              i <= '0;
            end
          end
        end
      end
    endcase
  end
end
```

**Listing 3:** Writing process.

32'hFFFFFFFF must be written to address 32'hFFC to toggle the programming mode off and return the LFSM into FSM operating mode.

A reset signal will zero all the bits in the LUT, and also set the counters **i** and **j** to zero. Therefore, the LUT must be rewritten when a reset has occurred.

Having the correct word at the corresponding position in the LUT is crucial for correct behavior of the FSM. This poses a challenge in the writing process. The writing must start at position (current state value 0, input value 0) and iterate using the counters **i** and **j** in Listing 3. Each word written is a line from the state transition table. Thus, care must be taken so that the state transition table is correct and will correspond to the correct position in the LUT. The transition table must have a word for every position, even if the position (a combination of state and inputs) is an unreachable state or not

used at all. Having to write all positions can be seen in relation to the LUT size issue. Having more clever indexing and writing by the bus may open possibilities to reduce the LUT size, which should be considered for future work.

**State register and reading the LUT** The state register in the LFSM is not different from what a was presented in Section 2.2.3. It holds the current state. The state register is declared as an *always_ff* block as shown in Listing 4. At each clock cycle, the state register will either remain in the current state or get a new state from the LUT. If the reset signal goes high, the state register will be set to zero.

```
always_ff @(posedge clk or posedge arst) begin
  // Default state is 0 on reset
  if (arst) begin
    state <= '0;
  end
  else begin
    state <= nextState;
  end
end
```

**Listing 4:** State register in LFSM.

Finding the correct output values and the next state is relatively straight forward once the writing of the configuration is finished. Listing 5 show how the LFSM finds the next state and the outputs. Since it is a combinatorial process, it is declared as an *always_comb*. As mentioned in Section 2.2.2, a Moore machine only need the current state to determine the outputs, while a Mealy machine needs both the state and the inputs. Thus, a parameter can be set when instantiating the LFSM whether or not it should be a Mealy or Moore machine.

As seen in Listing 5, the programming mode register is checked, which means the LFSM only sets output and next state when it is not in programming mode. In a Moore machine, input value zero is used constantly for finding the output. The use of input value zero is arbitrary. For a Mealy machine, the actual input value instead of zero. Using a Moore machine should make possibilities for reducing the area of the LUT after synthesis, while a Mealy machine opens possibilities for more advanced state machines.

**How to program the LFSM** As mentioned, making sure the right word is at the right position is crucial for correct behavior by the LFSM. The LFSM use a state transition table in the LUT. State transition tables is presented Section 2.2. For the current LFSM design, the user of the LFSM must manually create the state transition table for the FSM which is to be used. The state transition table must be binary coded. The state transition table is, as mentioned, written to the LUT using a bus interface.

```
if (MEALY == 1) begin : la_Mealy
  always_comb begin
    if(!programming_mode) begin //Only provide output and
     ↪  nextState while not in programming mode
      outputs = state_table [state] [inputs] [MAX_NUM_OUTPUTS +
       ↪  MAX_NUM_STATES - 1 : MAX_NUM_STATES]; //Only the
       ↪  outputs from LUT word
      nextState = state_table [state] [inputs] [MAX_NUM_STATES -
       ↪  1 : 0]; //Only the nextState bits
    end
    else begin
      outputs =  '0; //all outputs are zero when there is no
       ↪  valid program in the FSM
      nextState = '0; //nextState is default zero when no valid
       ↪  program
    end
  end
end
else begin : la_Moore
  always_comb begin
    if(!programming_mode) begin
      outputs = state_table [state] [0] [MAX_NUM_OUTPUTS +
       ↪  MAX_NUM_STATES - 1 : MAX_NUM_STATES]; //Outputs only
       ↪  determined by state. The use of input 0 is arbitrary.
      nextState = state_table [state] [inputs] [MAX_NUM_STATES -
       ↪  1 : 0];
    end
    else begin
      outputs =  '0;
      nextState = '0;
    end
end
end
```

**Listing 5:** Finding the next state and the outputs.

Creating the state transition table for the format for writing over the bus has so far been done by using a spreadsheet program like Microsoft Excel. The spreadsheet of the state transition table for the USB-FSM can be seen in Appendix B. In Microsoft Excel, it is possible to copy columns. So, the columns for the next state and the outputs are copied into a .txt-file and all spaces are removed. A simple Python script is used to parse this .txt-file into the format needed for the software or bus driver to write to the LUT. The script is added in Appendix C.

This way of programming the LFSM is not particularly user-friendly and mistakes can easily happen. Especially, creating the binary encoded state transition table is prone to mistakes. It is also difficult to find mistakes and debug the program. The amount of work to create the tables is also very dependent on the number of inputs, outputs, and states. During the work of this thesis, more emphasis was put into creating a functional LFSM rather than the programming interface, and it is left for future work to create a more user-friendly interface. Some ideas for refinement are for example to create a C-program for the FSM and then have a script parse the C-file and automatically create the format required for the LUT. An even more advanced solution which might be more user-friendly would be a graphical user interface where the FSM can be drawn as a state diagram and

a program can interpret the drawing and generate the format for the LUT.

## 4.2 Software based UPFSM

Using a microprocessor as an FSM is not a new idea. As presented in Section 3.3, it was considered already in 1987 [6], and ARM has a white paper [5] on the subject. The main inspiration for the software-based design is from the ARM white paper. Besides, implementing an FSM in software for running on a microprocessor opens possibilities for comparison and evaluation to the LFSM. The software-based UPFSM is for simplicity's sake called the SWFSM. Figure 4.5 show a simplified overview of the SWFSM design.

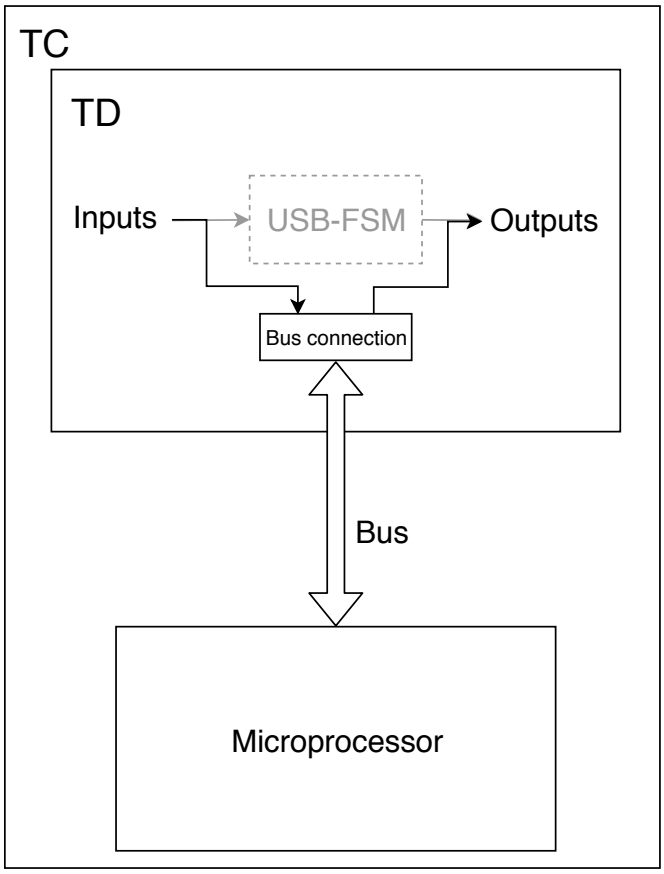



**Figure 4.5:** Simplified overview of how the SWFSM works with TD and TC. The USB-FSM is gray to illustrate how the signals are re-routed.

The SWFSM should have a microprocessor connected to the inputs and outputs from the original FSM. Since the microprocessor has multiple connections, a bus infrastructure is needed to communicate to all the various units connected to the microprocessor. Section 4.2.1 describes how the SWFSM was investigated an developed, and Section 4.2.2 describes the design that was implemented.

### 4.2.1 Development

The ARM white paper [5] was used as a starting point. As the existing TC was used in this thesis, a natural starting point. TC has a bus structure and a microprocessor, as well as frameworks for testing and developing software. From the ARM white paper, it was clear that some registers for reading and writing the FSM inputs and outputs were needed. These registers had to be connected to the main bus structure of TC so that the microprocessor could access them from software. It was decided to use one register for the inputs to the FSM and one register for the outputs.

As TC is an existing SoC from Nordic Semiconductor, and the SWFSM is in the cross-section of hardware and software design, cooperation with engineers from Nordic Semiconductor was necessary for the implementation. The development started at two ends, with one end being the software side, and the other end the hardware side. Engineers with knowledge of the software and hardware sides assisted in their respective fields. Eventually, the ends were tied together, and the design of the actual SWFSM could be carried out.

### 4.2.2 Design

Figure 4.6 shows a simplified overview of the SWFSM and how it is placed in TD. It can be divided into two major parts, the hardware side with signals and registers, and the software side which runs in the microprocessor. Each side will be presented in the following paragraphs.

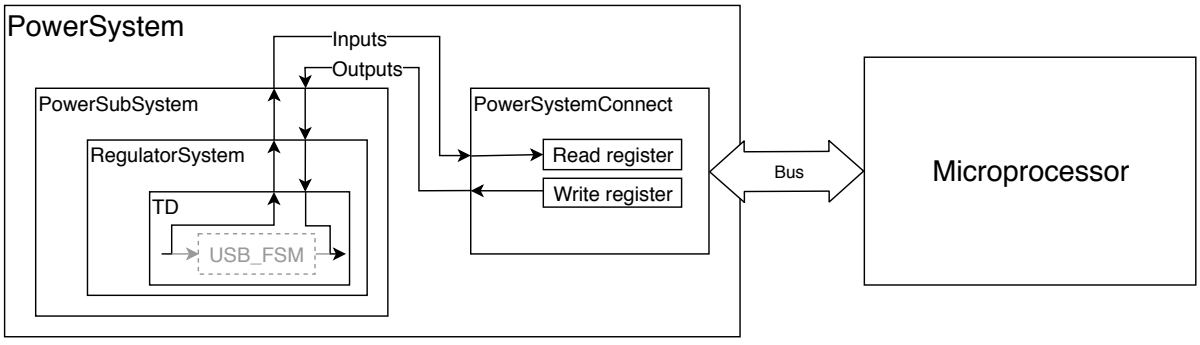



**Figure 4.6:** Overview of the SWFSM and its connections.

**Hardware side** The SWFSM is using registers and signals which are added to the hardware of TC. The existing USB-FSM in TD is removed, but the signals going in and out of the USB-FSM are routed to registers accessible to the microprocessor. TD is instantiated inside several layers of other IPs and subsystems. Thus, the input and

output signals had to be routed through all the layers to the module which is connected to the bus infrastructure.

As shown in Figure 4.6, TD is a part of a larger IP named PowerSystem. PowerSystem contains submodules PowerSubSystem and RegulatorSystem, which contains TD. However, PowerSystem is connected to the bus infrastructure via the submodule PowerSystemConnect. This means that the signals from TD must be routed to the top module, PowerSystem, and then into the PowerSystemConnect. PowerSystemConnect contains all the registers for all the modules in PowerSystem. The PowerSystem has a base address in the bus structure, and offsets are used to access the internal modules. The registers used for the USB-FSM did not exist in TC from before, so they had to be added with help from hardware-engineers from Nordic Semiconductor. The register for the input signals is made as a read-only register since these signals originate from logic in TD and the microprocessor does not need to write to them. The register for the output signals is made as a write-only register, as these signals used to be the outputs of the original USB-FSM. An existing testbench for TC was used to verify that the signals were added correctly.

**Software side** With help from software engineers at Nordic Semiconductor, the registers implemented in hardware is added to the firmware for TC. The framework is built up around software header-files containing structs. These structs are used to break down the address which is used on the bus. As mentioned, each module has a base address and an offset. The base address is used to find the correct struct, and the offset is specified in the struct.

The program for running the SWFSM is written in C and is shown in Listing 6. As TC is a complex SoC, unnecessary code from an existing C-program was removed and replaced with the code for running the SWFSM. The existing C-program already had a framework for co-simulating the hardware and software, which reduced the complexity and time needed to implement the SWFSM.

The code is not very different from the SystemVerilog code used for the hardwired USB-FSM and is based on the state transition diagram shown in Figure 3.1. The program uses an infinite while-loop with a switch case. The states are stored as an enum variable for readability. Each loop, the register "POWER_SYSTEM->FSM_INPUTS" is polled and stored as the variable "temp", and the switch statement checks the current state. Based on the current state, the output is written to the register "POWER_SYSTEM->FSM_OUTPUTS". The writing of output is hexadecimal in the C program, but it corresponds to the binary output for that state. In each state, an if-else statement is used to compute the next state based on the polled inputs. The comments in the code indicate which bits in the input register should be checked in each state. A bitwise "AND" and a shift right operation is used to mask out the don't care-bits.

```c
int main(void)
{
  volatile uint32_t temp;
  uint32_t counter = 0;

  enum FSMSTATE {STATE_0, STATE_1, STATE_2, STATE_3, STATE_4}
   ↪  state;

  state = STATE_0; // Start
  POWER_SYSTEM->FSM_OUTPUTS = 0;

  while(1){
    counter++;
    temp =  POWER_SYSTEM->FSM_INPUTS;

    //Trigger input_0 backdoor
    if(counter == 1) *(volatile uint32_t *)0x500045F8 = 0x3;

    // Trigger input_2 backdoor
    if(counter == 5) *(volatile uint32_t *)0x500045E8 = 0x3;

    if(counter > 1){
    switch (state){
      case (STATE_1) :
          POWER_SYSTEM->FSM_OUTPUTS = 4; //100
          if ((temp & 8)>>3 == 0) state = STATE_0; //!input_0 0xxx
          else if (((temp & 8)>>3  == 2) && ((temp & 4)>>2 == 0))
           ↪  state = STATE_1; //input_0 and !input_1 10xx
          else if (((temp & 8)>>3  == 1) && ((temp & 4)>>2 == 1))
           ↪  state = STATE_2; //input_0 and input_1 11xx
        break;
      case (STATE_2) :
          POWER_SYSTEM->FSM_OUTPUTS = 5; //110
          if ((temp & 8)>>3 == 0) state = STATE_0; //!input_0 0xxx
          else if ((temp & 2)>>1 == 0) state = STATE_2; //!input_2
           ↪  xx0x
          else if ((temp & 2)>>1 == 1) state = STATE_3; //input_2
           ↪  xx1x
        break;
      case (STATE_3) :
          POWER_SYSTEM->FSM_OUTPUTS = 6; //110
          if ((temp & 8)>>3 == 0) state = STATE_0; //!input_0 0xxx
          else if ((temp & 2)>>1 == 0) state = STATE_2; //!input_2
           ↪  xx0x
          else if ((temp & 1) == 0) state = STATE_3; //!input_3
           ↪  xxx0
          else if ((temp & 1) == 1) state = STATE_4; //input_3
           ↪  xxx1
        break;
      case (STATE_4) :
          POWER_SYSTEM->FSM_OUTPUTS = 7; //111
          if ((temp & 8)>>3 == 0) state = STATE_0; //!input_0 0xxx
          else if ((temp & 2)>>1 == 0) state = STATE_2; //!input_2
           ↪  xx0x
          else if (((temp & 8)>>3 == 1) && ((temp & 2)>>1 == 1))
           ↪  state = STATE_4; //input_0 and input_2 1x1x
        break;
      default : // case (STATE_0)
          POWER_SYSTEM->FSM_OUTPUTS = 0;
          if ((temp & 8)>>3  == 0) state = STATE_0; //!input_0
           ↪  0xxx
          else if ((temp & 8)>>3 == 1) state = STATE_1; //input_0
           ↪  1xxx
        break;
    } //Switch
    } //if counter > 1
  } // end while
  return 0;
}
```

**Listing 6:** Main function from C program for the SWFSM. The counter is used for testing purposes, described in Section 5.2.

# 5 Testing, results, and discussion for the UPFSM

This section will present how the two UPFSM solutions were tested and verified. The results from the LFSM and the SWFSM will be presented and briefly discussed. A thorough evaluation and discussion will be conducted in Section 6.

## 5.1 LFSM

Testing was performed continuously during the development of the LFSM. At first, a configuration for a simple FSM like the example FSM from Section 2.2 was created and used. The behavior was verified by examining waveforms in the simulation tool QuestaSim. The LFSM was instantiated in TD and replaced the original hardwired USB-FSM. As an IP-level testbench for verification of TD already existed, the behavior of the LFSM was functionally verified using this testbench. The testbench needed the signals for the current and next state from the LFSM, so they had to be extended as outputs. These signals were used in asserting the correct behavior. After some attempts, the LFSM passed the TD testbench without errors. A sanity check was also performed, by deliberately adding errors in the FSM configuration, and the testbench did not pass accordingly.

However, the LFSM lacks complete verification. An IP-level testbench for the LFSM should have been made. This would include the use of SystemVerilog assertions and randomized stimulus. However, it is difficult to properly verify the behavior since it depends on the FSM configuration. A complete verification is left for future work.

### 5.1.1 Results from synthesis

Synthesis was carried out to find the area and number of flip-flops needed for the LFSM and TD. The LFSM was synthesized with different parameters and sizes for the LUT, which included Mealy versus Moore. TD was synthesized both with and without the LFSM. As different parameters could be set, such as Moore versus Mealy, and different LUT sizes, several synthesis runs were conducted. The results are shown in the following figures.

Looking at the number of flip-flops from Figure 5.1, it is easy to see that TD with LFSM is substantially larger than TD with the original hardwired FSM. The functionality is the same between the versions. The Moore machine version has over 4 times as many flip-flops as the original, and the Mealy machine version has almost 7 times as many. Increase in the number of flip-flops also increase the number of transistors and thus the area, which is can be seen from Figure 5.2. The figure has the area relative to the original TD, which is set to 1. It also shows the distribution between combinatorial and noncombinatorial

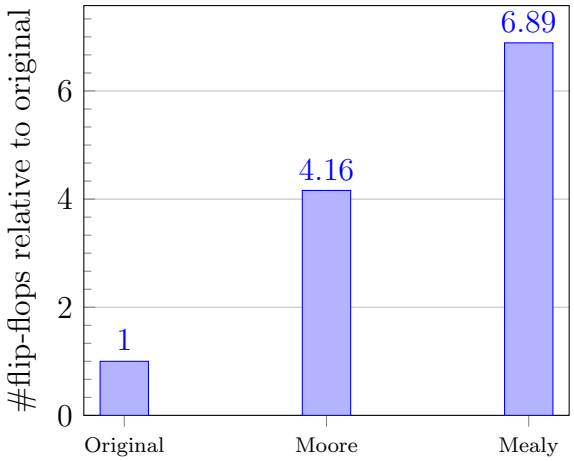



**Figure 5.1:** Number of flip-flops relative to original TD.

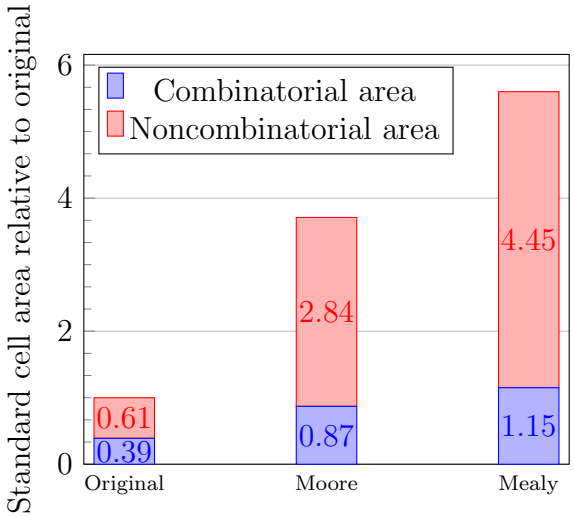



**Figure 5.2:** Area relative to original TD.

area. The increase in area is mostly caused by an increase in noncombinatorial area. Noncombinatorial area means sequential logic, such as registers. The increase is thus expected, as the LUT is made from registers. The increase in combinatorial logic comes from the logic used to write and read the LUT, such as multiplexers and decoders.

Figure 5.3 show the number of flip-flops for the LFSM for different parameter sizes and for Mealy versus Moore versions. As mentioned in Section 2.5, the LUT size is determined by the number of input and output signals, as well as the number of bits needed to represent the state set. The increase in the number of flip-flops is exponential, and this can be seen in Figure 5.3. The synthesis with 4 input signals, 3 state-bits, and 3 output signals reflect the size used for the USB-FSM. The version with 6 input signals, 6 state-bits, and 6 output signals is at the limit of what the synthesis tool could handle, and the time needed for synthesis was much longer than for the smaller versions. Synthesis with fewer inputs

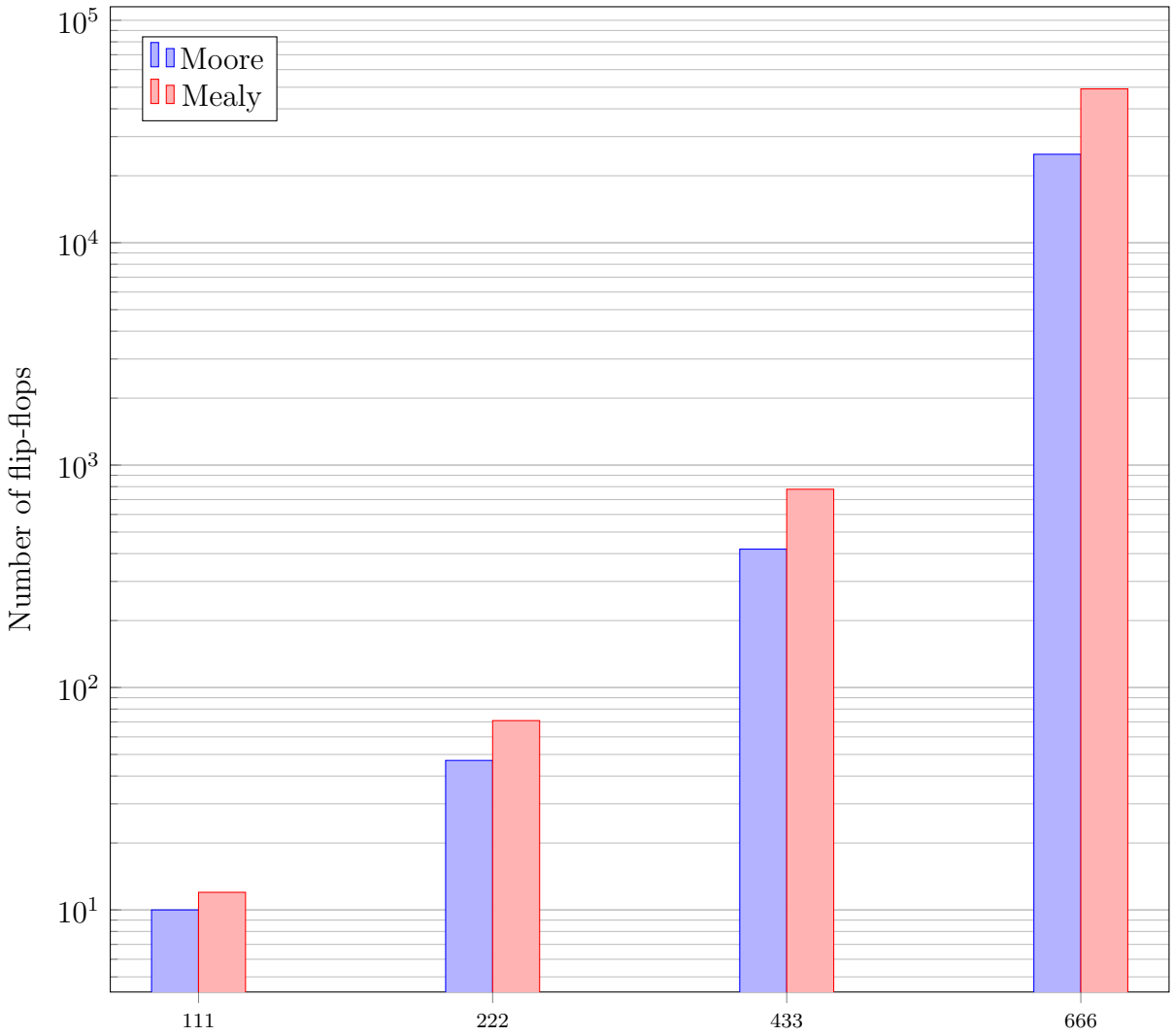



**Figure 5.3:** Number of flip-flops for different LUT sizes of LFSM. X-axis numbers are {#input width, #state-bits, #output witdth}, Y-axis is logarithmic with base 10.

and outputs, but more state-bits could have been performed, but a ballpark estimate can be done by using the LUT size from Equation (5).

The difference in area can be seen in Figure 5.4. The difference between Moore and Mealy machines comes from the fact that only the state is needed to compute the outputs. Thus, the LUT can almost be half the size. The synthesis tool is able to optimize the LUT based on how many flip-flops that are not in use, and for a Moore machine version, this would be almost half the registers. As seen in Listing 5 from Section 4.1.2, in the LFSM, input value zero is constantly used to provide the output in a Moore instantiation, and the rest of the flip-flops will never be used.

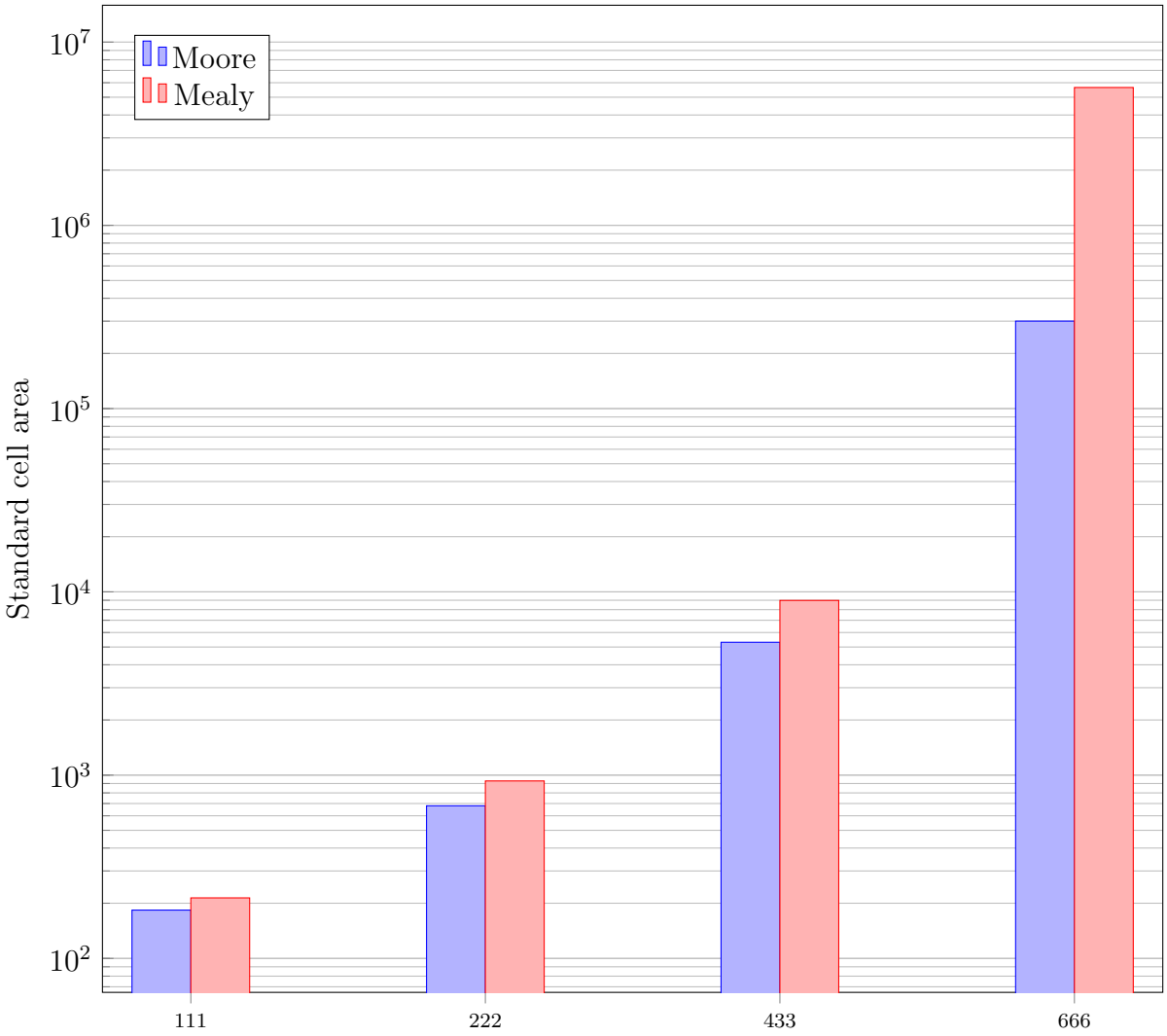



**Figure 5.4:** Area of LFSM. X-axis numbers are {#inputs, #states, #outputs}, Y-axis is logarithmic with base 10.

## 5.2 SWFSM

The SWFSM is tested by simulating TC in QuestaSim® with the SWFSM program running on the microprocessor. Only an RTL-model of TC is used, so no physical design was used. As there was not enough time to create a proper testbench with test-stimulus for the SWFSM, backdoor-signals to TD was used instead. Backdoor-signals are signals that can be triggered by writing to registers from the C-program. By triggering these signals, the USB regulator is enabled and another signal can emulate that a USB cable is connected. They will thus act as input signals to the SWFSM. The backdoor-signals are set by adding a counter in the while-loop in the SWFSM code, which can be seen in Listing 6. When the counter reaches a value, the signal is set. However, backdoor signals only exist for the signal "input_0" and "input_2". Triggering "input_0" automatically triggers "input_1" after one TD clock cycle. The input signal "input_3" cannot be backdoor-triggered since

a backdoor-signal does not exist in the design. The lack of this backdoor-signal means the entire SWFSM cannot be properly tested and verified for all the states. However, propagating through the states "STATE_0", "STATE_1", "STATE_2, and "STATE_3 can be done, which gives a good indication of it working. A ballpark estimate of the speed can be also be made.

An interesting test would be to see how fast the microprocessor is able to compute an output following a change in the inputs. This is measured by the number of clock cycles the microprocessor uses. The measurement is done using QuestaSim waveform generator. By finding the time interval between two events in the waveform and divide this time by the clock period, the number of clock cycles is found. The clock frequency for the microprocessor is eight times higher than the frequency for TD and the bus.

The number of clock cycles can be found for two intervals:

1. The number of cycles the microprocessor needs to read the input register and write a new output to the output register.
2. The number of cycles needed from an input change in TD to the output change is present in TD. This includes the latency caused by the bus.

For the first interval, the microprocessor use 640 microprocessor clock cycles, or 80 TD/bus-cycles. This is the worst case interval found in the waveform.

The second interval needed 992 microprocessor cycles, or 124 bus/TD cycles.

Both these intervals indicate that the SWFSM is very slow compared to the original TD. The original TD will provide an output every bus/TD clock cycle, whilst SWFSM will provide an output every 124th cycle. Note that this is only for one of the state transitions, and might not even be a worst-case scenario.

One explanation can be found in the C-program from Listing 6. The program runs in a loop which first read the input registers. Then, it enters the switch statement and the current state. Here, the output is set for the current state, and then the next state is computed. Then, the loop starts over again, by reading the input. Henceforth, it now enters the case for the state computed in the previous loop, where the outputs are set. In this manner, when the input is changed, the program reads the inputs two times before the output is set. This makes it slower, and the program should be optimized to avoid this. According to [5], a possible way to optimize the code is to use "goto" with labels. The "switch" statement will not be needed, and this should make the program more efficient because it does not need to jump back to the "switch" statement in each transition.

Another possible explanation was found after discussion with Nordic Semiconductor engineers. It turned out that the SWFSM software was running from flash memory. This means the flash will add significant delay. Enabling the use of a cache would most likely

reduce the number of microprocessor cycles significantly since a cache is a faster memory than flash. Another option would be to run the program from RAM, which would also be faster than flash memory. Unfortunately, these possibilities were discovered late, and it was not enough time to optimize this.

Areawise, synthesis for TD without a USB-FM show that it is smaller than the original TD and also the LFSM version. As the USB-FSM is removed from TD, the area is reduced. The registers for inputs and outputs, which are not in TD, must be taken into consideration. But they will at most be 32 flip-flops each, depending on the number of inputs and outputs. The area is also not dependent on FSM state, except for the input and output registers. In addition, the program will need space in the memory of TC, which is also an area penalty.

# 6 Evaluation and discussion

This section presents a comparison and evaluation of the three different methods to implement an FSM. They are compared and evaluated in terms of area, throughput, power, complexity, and ease of use. A small summary of the findings is also given.

## 6.1 Area

Having a small area as possible is very useful for IC and SoC designs. Small area means lower power consumption, less manufacturing cost, and smaller physical size.

The SWFSM is the smallest in area. This is expected, as no logic is needed in hardware when it is all performed in software. The LFSM is the largest. It is, however, not very large for small FSM with few inputs, outputs, and states. But, if either of these parameters is increased, the area will increase exponentially. This can be seen from Equation (5). When synthesis was performed, the synthesis tool was not able to synthesize larger than 6 inputs/outputs signals and 6 state-bits. The LFSM is best suited for small FSMs. The FSMs should also be Moore machines, as this nearly cuts the LUT size in half.

A possibility for reducing the LUT size is to use RAM blocks instead of registers. This will remove the area occupied by the LUT and use RAM blocks that typically are found in SoCs. Some logic and wires are needed to access the RAM which costs some area.

Having a hardwired FSM will cost some area, and the area will also depend on the number of states, inputs, and outputs. But it is less area consuming than the LFSM. In terms of area, the SWFSM is better, at least when there are many inputs, outputs, and states.

## 6.2 Throughput and speed

Having a fast FSM is not always needed, but in many cases, a constraint exists on the speed of the FSM.

**Hardwired FSM** A hardwired FSM has combinatorial logic to compute the next state and the outputs, and a sequential register to store the next state. This means it can provide output every clock cycle. Some delay comes from the critical path in the digital circuit, but nothing excessive compared to the rest of the circuit.

**LFSM** The LFSM also provide output every clock cycle. Next state and output are generated by decoding what is present at a position in the LUT, and the next state is stored

in a register. However, the LFSM must be programmed before it behaves as an FSM, and this programming takes time. The LFSM requires one clock cycle per position in the LUT for writing the program, in addition to two cycles to toggle the ProgrammingMode register. If a large LUT is needed, such as the one synthesized with 6 input signals, 6 output signals and 6 state.bits, it will use $2^{6+6} = 4096$ clock cycles to fill all the positions in the LUT. It must also be reprogrammed after a system reset, and this must be taken into account when using it.

**SWFSM** The SWFSM is very slow compared to the other solutions. As mentioned in Section 5.2, it takes 124 bus/TD clock cycles to compute an output from an input change, whereas the hardwired FSM and the LFSM only use one clock cycle.

The SWFSM will most likely share the microprocessor with several other programs and processes. This means additional time is needed to schedule the FSM tasks. An interrupt can be used, but this also adds a delay as the microprocessor must store what it is doing before it can handle the interrupt. The program for the SWFSM is not very optimized, and the SWFSM could probably operate faster if more time and thought is put into optimizing the code. Also, including the changes proposed by Nordic Semiconductor engineers would also make significantly SWFSM faster.

If strict timing control and fast operation are required, it is probably better to avoid using a microprocessor. The LFSM is equally fast as a hardwired when it is finished programming, but will require substantial amounts of time if the FSM has many states, inputs, and outputs.

## 6.3 Power consumption

Not too much emphasis has been put into power consumption for the SWFSM and the LFSM. However, low power consumption is very important in SoCs and ICs, which typically run on battery power. Power consumption is related to the area and speed of the design.

The LFSM is the most area-dependent and use registers for storing the LUT. The LUT is not clock gated by design, but modern synthesis tools are able to insert clock gating automatically, which helps to reduce the power consumption.

The SWFSM runs on a microprocessor, and the power consumption is related to the power consumption of the microprocessor. The program currently used for the SWFSM is causing the microprocessor to stay active and polling registers. This consumes a lot of power. Using interrupts instead would allow the microprocessor to go to sleep when there is no activity on the input registers. Sleeping the microprocessor obviously reduce

the power consumption dramatically. How often it must be awakened will depend on the FSM which is used and how active that FSM is. Also, if the number of clock cycles is reduced, this would also save power since the microprocessor can return to sleep faster.

## 6.4 Complexity and ease of use

Having a UPFSM will most likely always come with some drawbacks and added complexity. A programmable system will need extra features such as a programming interface and a memory to store the program. Hardwired FSMs are not more complicated than they need to be.

**LFSM** The LFSM is not very complicated, at least not for small FSMs. As mentioned, the area increases exponentially with the size of the LUT, and, likewise, the complexity increases. In terms of ease of use, the LFSM is relatively straight forward to use in a design, as it is a standalone IP with parameterized inputs and outputs. A designer only needs to specify the number of bits needed in the parameters and have a bus interface present and hook it up accordingly. The real problems arrive when it is time to program it.

During the work of this thesis, the writing over the bus has only been done in a testbench with a bus driver. For a real application, software code must be written which writes to the LUT after each reset. Creating the program is not very user-friendly either, as a state transition table must be created. Creating such tables manually is not very difficult for small FSMs, but the complexity increases exponentially for large FSMs. The state transition table for the USB-FSM, which can be seen in Appendix B, has 80 rows and 15 columns. If a table had been made in the same manner for an FSM with 6 inputs/outputs and 6 state bits, the table would have 384 lines and 26 columns. This is much more prone to mistakes, and it is challenging to find the mistake.

Having a system that can generate the software code automatically, by for instance creating the state transition table from a C-program like the program used in the SWFSM, would make the LFSM much more user-friendly, even with large FSMs.

**SWFSM** The SWFSM may be more user-friendly than the LFSM, at least considering the programming interface. For a basic implementation, only a simple C program is needed. An SoC will have all the necessary framework and the toolchain needed to write programs to the microprocessor. More specialized and optimized code can also be written, which can reduce the power consumption and memory required.

The SWFSM will, however, need the input and output signals to be read and writable by the microprocessor. If the FSM resides deep in several layers of IPs and modules, the complexity increase as the signals must most likely be forwarded through all the IP layers to a module that is connected to the bus in the SoC. An SWFSM will also occupy bus address space, which is not unlimited. If the microprocessor would run several FSMs, this will also increase the complexity.

The SWFSM has an advantage over the LFSM as the complexity is not increased as much for larger FSMs. Albeit, the software program is larger, but the general ideas are not much different. Having a software program allows for the use of debuggers and other tools which help find bugs and faults in the program.

**Hardwired FSM** A hardwired FSM is less complicated than a UPFSM since it does not need the extra features needed for the programmability. However, it is less flexible and not universal. A custom FSM must be made for every design. Depending on the user, it might be less user-friendly. A hardware engineer might find it easier to implement a hardwired FSM than an SWFSM, while a software engineer might find it easier to implement an SWFSM.

## 6.5 Summary and future work

The discussion and comparison presented is summed up in Table 6.1.

It is difficult to say whether one type of FSM implementation is better than the other. Compromises must be made anyway. As mentioned, the hardwired FSM is not programmable, and it, therefore, has the disadvantage of not being able to change behavior after manufacturing. Both the UPFSM implementations are programmable, but are more complex and cost more in terms of area or speed. Even though this thesis aimed to create a UPFSM, such a UPFSM might only be worth it if the programmability is strictly required. The SWFSM has an advantage in area and ease of programmability over the LFSM when it comes to large FSMs, while the speed of the LFSM is better than the SWFSM for smaller FSMs.

Albeit, the ideas presented in this thesis might be a good starting point for further developing a UPFSM. The LFSM has potential if solutions to reduce its area and make it easier to use can be found. Ideas for area reduction are more clever storage in the LUT, which takes unreachable states into account. To make it easier to program, an automatic system can be developed which reads an FSM program written in C or a different, higher-level programming language and creates the format needed in the LFSM.

The SWFSM also has potential if the speed constraints for the FSM are low. A starting

**Table 6.1:** Summary and comparison of the three types of FSMs considered

| | Hardwired FSM | LFSM | SWFSM |
|---|---|---|---|
| Programmable | No | Yes | Yes |
| Area | Small area, but depends on FSM size. | Large area, at least for large FSMs. Mealy machines require more area than Moore machines. | Small area, and FSM size has little influence on area. |
| Throughput and speed | Fast. Output and next state computed in every clock cycle. | As fast as a hardwired FSM when it is programmed. Programming takes time, which lead to overall slower speed. | Slow. Speed depends on the microprocessor speed and bus clock frequencies. |
| Power consumption | Low power consumption. Small area and few registers needed. Clock gating can be effective if state is seldomly changed. | High power consumption. Many registers and large area is required, but it depends on FSM size. | Low power consumption. The microprocessor has high power consumption, but can usually be put to sleep. |
| Complexity and ease of use | Low complexity and easy to use. Takes time to implement custom for every application. Possibly easy to use for HW engineers, and more difficult for SW engineers. | Complexity depends heavily on the FSM size. Easy to implement, but difficult to program. Requires bus interface and a microprocessor to program it. | Low additional complexity if a microprocessor and bus exists. Added complexity if signals must be forwarded. Relatively easy to program, for almost any FSM size. |

point to further develop the SWFSM, is to improve the program code and run the software with cache enabled or run it from RAM. Avoiding polling the registers in a relatively slow while-loop could prove useful. Also, having sufficient control of the timing of the microprocessor and bus clocks would most likely increase the speed. However, the implementation of an SWFSM in an already existing SoC is not easy either, so if an SWFSM is to be used, it is probably a good idea to develop the SoC with SWFSMs in mind. This would involve easier connections to the bus infrastructure and a proactive design for what FSMs should be partitioned to software in the microprocessor.

A different possibility for future work is to use the findings in this thesis to develop a custom processor architecture such as in [22] and [23]. A custom architecture can potentially avoid the vast area which the LFSM use, and be more user-friendly. The speed would also most likely be higher, as it can operate separately from the microprocessor. A custom architecture can even be implemented as an accelerator for the microprocessor, which offloads the FSM tasks from the microprocessor.

# 7 Conclusion

State machines are used extensively in digital hardware designs. The FSMs are usually hardwired, and their behavior cannot be changed after the IC or SoC product has been manufactured. Having a programmable FSM would allow this. Therefore, this thesis has proposed two different solutions to make a UPFSM; the LFSM and the SWFSM.

The LFSM is designed as an IP which can be used to replace existing FSMs. It is based on a programmable LUT, which can be programmed using a bus interface. This solution is as fast as a hardwired FSM in terms of speed, but sacrifices area and programming complexity. It can be instantiated as a Moore or a Mealy machine, and the Moore machine has substantially less area. For comparison, it is used to replace an FSM in a power regulator controller, with 4 inputs, 3 outputs, and 3 bits to represent the state set. Synthesis of the LFSM in a Moore machine configuration returned 3.71 times more area than the original, and a Mealy machine configuration returned 5.6 times more area. The area increases exponentially for larger FSMs with more inputs, outputs, and state bits. A 6 inputs, 6 output, and 6 state bits LFSM will have an area approximately 57 times (Moore) or 63 times (Mealy) larger than a 4 input, 3 output, and 3 state bits FSM. The programming complexity also increases with FSM size, since the LFSM need a full state transition table for all combinations of inputs and current state.

The SWFSM is a software-based solution running on a microprocessor. The input and output signals for the FSM is forwarded to registers that are read and written to by the microprocessor. The microprocessor runs a program for the FSM. This proved to have a reduction in area, but a much slower operating speed. It takes 992 microprocessor clock cycles from when a change happens in the input signals to a change can be seen in the output signals. However, it is easier to use for larger FSMs.

Having a programmable FSM will have some drawbacks and will always be a compromise. A hardwired FSM is probably the simplest solution, but if programmability is required, the LFSM or the SWFSM are both good solutions. Which one to choose depends on the FSM size, speed constraints, and area constraints.

# References


[1] L. Colombus, *"2018 Roundup Of Internet Of Things Forecasts And Market Estimates"*. Forbes Magazine. [Online]. Available: https://www.forbes.com/sites/louiscolumbus/2018/12/13/2018-roundup-of-internet-of-things-forecasts-and-market-estimates (visited on May 19, 2019).

[2] K. L. Lueth, *"State of the IoT 2018: Number of IoT devices now at 7B – Market accelerating"*. IoT Analytics. [Online]. Available: https://iot-analytics.com/state-of-the-iot-update-q1-q2-2018-number-of-iot-devices-now-7b/ (visited on May 20, 2019).

[3] T. Simonite, *"Moore´s Law Is Dead. Now What?"*. MIT Technology Review. [Online]. Available: https://www.technologyreview.com/s/601441/moores-law-is-dead-now-what/ (visited on May 19, 2019).

[4] D. A. Patterson and J. L. Hennessy, *Computer organization and design : the hardware/software interface*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann, 2014.

[5] J. Yiu, "Software based Finite State Machine (FSM) with general purpose processors," ARM, Tech. Rep., Jan. 2013.

[6] D. J. Comer, "Digital System Design: State Machine Versus Microprocessor Controller," *IEEE Transactions on Education*, vol. E-30, pp. 102–106, 2 May 1987.

[7] Wikipedia.org, *"Semiconductor intellectual property core"*. Wikipedia.org. [Online]. Available: https://en.wikipedia.org/wiki/Semiconductor%20_intellectual_property_core (visited on May 28, 2019).

[8] A. D. Aziz, J. Cackler, and R. Young, *Automata theory*, Sep. 2004. [Online]. Available: https://cs.stanford.edu/people/eroberts/courses/soco/projects/2004-05/automata-theory/basics.html.

[9] A. Jantsch, *Modeling Embedded Systems and SoC's: Concurrency an Time in Models of Computation*. San Francisco, CA, USA: Morgan Kaufmann Publishers, 2004.

[10] M. Sipser, *Introduction to the Theory of Computation*, 2nd ed. Boston, MA, USA: Thomson Course Technology, 2006.

[11] J. Loeckx, *Computability and Decidability: An Introduction for Students of Computer Science*, 1st ed. Berlin, Germany: Springer-Verlag, 1972.

[12] J. E. Hopcroft, R. Motwani, and J. D. Ullmann, *Introduction to Automata Theory, Languages and Computation*, 2nd ed. Boston, MA, USA: Addison-Wesley, 2001.

[13] E. F. Moore, "Gedanken-experiments on sequential machines," *Automata studies*, vol. 34, pp. 129–153, 1956.

[14] G. H. Mealy, "A method for synthesizing sequential circuits," *The Bell System Technical Journal*, vol. 34, no. 5, pp. 1045–1079, Sep. 1955.

[15] S. Sutherland, S. Davidmann, and P. Flake, *SystemVerilog for Design*, 2nd ed. New York, NY, USA: Springer, 2006.

[16] A. Osborne, *An Introduction to Microcomputers. Volume 1: Basic Concepts*, 1st ed. Berkeley, CA, USA: Sybex, 1976.

[17] E. O. Hwang, *Digital Logic and Microprocessor Design with VHDL*, 2nd. ed. Toronto, Ontario, Canada: Thomson, 2006.

[18] M. M. Mano and C. R. Kime, *Logic and Computer Design Fundamentals*, 4th. ed. Upper Saddle River, NJ, USA: Pearson, 2008.

[19] C. Bobda, *Introduction to Reconfigurable Computing - Architectures, Algorithms and Applications*, 1st ed. Doordrecht, Netherlands: Springer, 2007.

[20] Doulos ltd., *SystemVerilog Golden Reference Guide*. 2012.

[21] Mentor Graphics, *"Questa® Advanced Simulator"*. Mentor Grapics. [Online]. Available: `https://www.mentor.com/products/fv/questa/` (visited on Mar. 17, 2019).

[22] L. Wangyang and L. Chin-Tau, "Design of A Programmable State Machine Architecture for Packet Processing," *IEEE Micro*, vol. 23, pp. 32–42, 4 Aug. 2003.

[23] S. Hatta, N. Tanaka, and T. Sakamoto, "Area-efficient Programmable Finite-state Machine Toward Next Generation Access Network SoC," *SASIMI 2018 Proceedings*, pp. 8–13, 2018.

# Appendices

## A LFSM source code

```
module LFSM #(
  parameter ID_LFSM_RW_BASE = package_file::LFSM_BASE,
  parameter BUS_AW = package_file::BUS_AW,
  parameter BUS_DW = package_file::BUS_DW,
  parameter BUS_WW = package_file::BUS_WW,
  parameter MAX_NUM_STATE_BITS = package_file::MAX_NUM_STATE_BITS, //number of state bits
  parameter MAX_INPUT_W = package_file::MAX_INPUT_W, //number of input bits
  parameter MAX_OUTPUT_W = package_file::MAX_OUTPUT_W, //number of output bits
  parameter MEALY = package_file::MEALY
  )(
  input  logic                                ck,
  input  logic                                arst,
  input  logic  [MAX_INPUT_W-1:0]     inputs,
  output logic  [MAX_OUTPUT_W-1:0]    outputs,
  output logic  [MAX_NUM_STATE_BITS-1:0] state,
  output logic  [MAX_NUM_STATE_BITS-1:0] nextState,

  // -- BUS interface inputs
  input  logic                                arst,
  input  logic                                ck,
  input  logic   [BUS_AW-1:0]                 bus_adress, // Bus address.
  input  logic   [BUS_WW-1:0]                 bus_data, // Bus data from microprocessor.
  input  logic                                bus_write_enable
  );

  logic programming_mode;
  logic [2**MAX_NUM_STATE_BITS-1:0] [2**MAX_INPUT_W-1:0] [MAX_OUTPUT_W+MAX_NUM_STATE_BITS-1:0] state_table;

  // Used for iterating through LUT
  logic [MAX_NUM_STATE_BITS-1:0] i;
  logic [MAX_INPUT_W-1:0] j;

  // Write LUT
  always_ff @(posedge ck or posedge arst) begin : la_BUSWrite
    // LUT set to zero on reset, and index variables are set to zero
    if (arst) begin
        state_table <= '0;
        i <= '0;
        j <= '0;
    end
    else begin
        case(bus_adress)
          32'hFFC : begin //Set FSM in programming mode by sending FFFFFFFF to address FFC
            if(bus_write_enable && bus_data == 32'hFFFFFFFF) begin
              if (programming_mode) programming_mode <= '0; //Toggle programming mode
              else programming_mode <= '1;
            end
          end
          //For every entry in state table to be indexable using inputs and state bits, the below must be done:
          ID_LFSM_RW_BASE : begin
              if(bus_write_enable) begin
                if(programming_mode) state_table[i][j] <= bus_data[MAX_OUTPUT_W+MAX_NUM_STATE_BITS-1:0];
                // Incrementing indexes
```

```
54                    j <= j + 1;
55                    if(j >= 2**MAX_INPUT_W-1) begin
56                      i <= i + 1;
57                      j <= '0;
58                      if(i >= 2**MAX_NUM_STATE_BITS-1) begin
59                        i <= '0;
60                      end
61                    end
62                  end
63              end
64            endcase
65      end
66    end
67
68    // State register
69    always_ff @(posedge ck or posedge arst) begin : la_state_reg
70      // Default state is 0 on reset
71      if (arst) begin
72        state <= '0;
73      end
74      else begin
75        state <= nextState;
76      end
77    end
78
79  // Output and nextState functions
80  if (MEALY == 1) begin : la_Mealy
81    always_comb begin
82        if(!programming_mode) begin //Only provide output and
          ↪  nextState while not in programming mode
83          outputs = state_table [state] [inputs]
            ↪  [MAX_OUTPUT_W+MAX_NUM_STATE_BITS-1:MAX_NUM_STATE_BITS];
            ↪  //Only the outputs
84          nextState = state_table [state] [inputs]
            ↪  [MAX_NUM_STATE_BITS-1:0]; //Only the nextState bits
85        end
86        else begin
87          outputs =  '0; //all outputs are zero when there is no
            ↪  valid program in the FSM
88          nextState = '0; //nextState is default zero when no valid
            ↪  program
89        end
90      end
91  end
92  else begin : la_Moore
93    always_comb begin
94        if(!programming_mode) begin //Only provide output and
          ↪  nextState while not in programming mode
95            outputs = state_table [state] [0]
                ↪  [MAX_OUTPUT_W+MAX_NUM_STATE_BITS-1:MAX_NUM_STATE_BITS];
                ↪  //Outputs only determined by state. The use of
                ↪  "input" 0 is arbitrary.
96            nextState = state_table [state] [inputs]
                ↪  [MAX_NUM_STATE_BITS-1:0];
97        end
98        else begin
99          outputs =  '0; //all outputs are zero when there is no
            ↪  valid program in the FSM
100         nextState = '0; //nextState is default zero when no valid
            ↪  program
101       end
102     end
103 end
104
105 endmodule
```

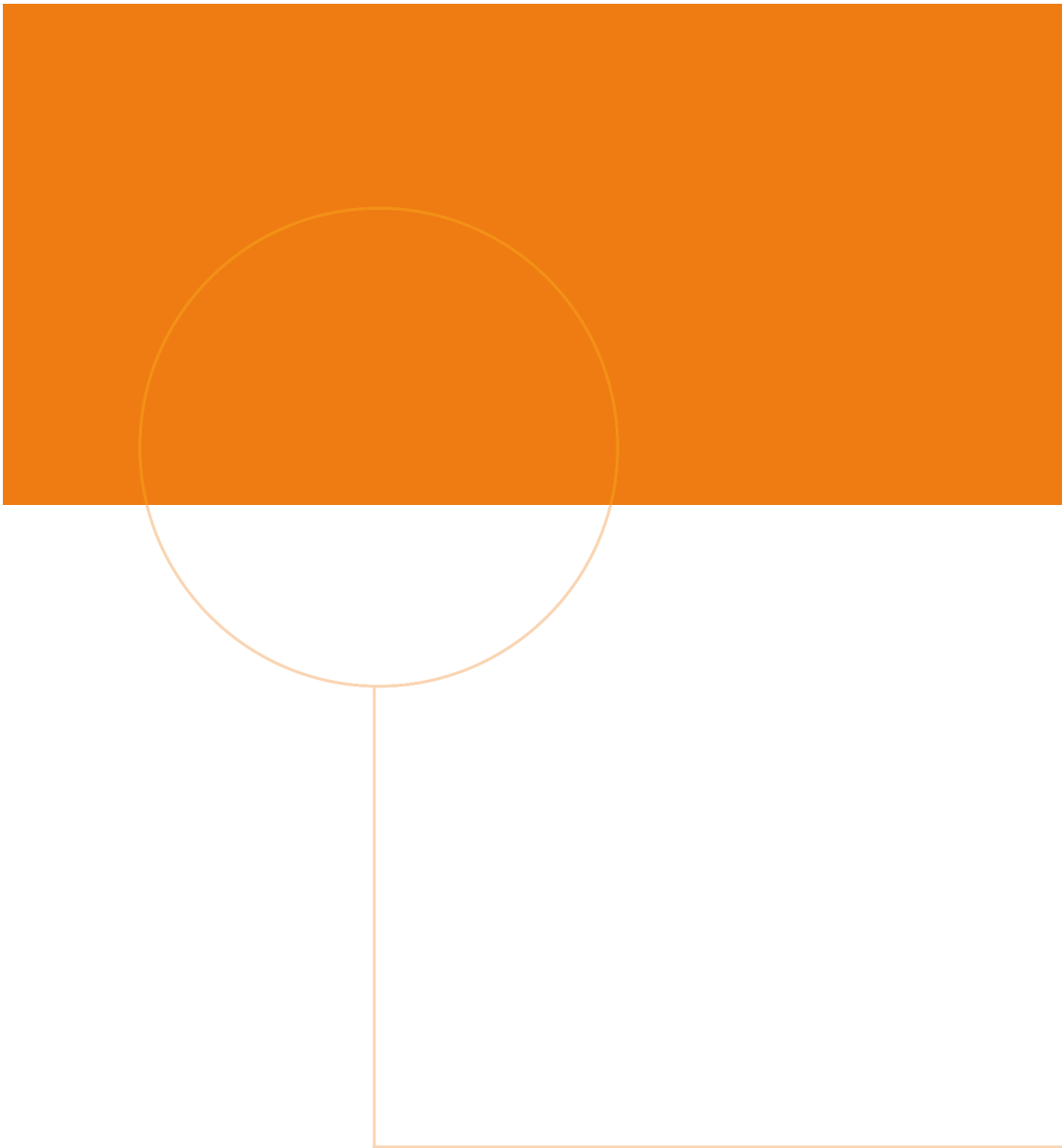
# B Spreadsheet for USB-FSM

| STATE | STATE_BITS | | | INPUTS | | | | OUTPUTS | | | NEXT_STATE | NEXT_STATE_BITS | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | input_0 | input_1 | input_2 | input_3 | output_0 | output_1 | output_2 | | | | |
| STATE_0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | STATE_0 | 0 | 0 | 0 |
| | | | | 0 | 0 | 0 | 1 | 0 | 0 | 0 | STATE_0 | 0 | 0 | 0 |
| | | | | 0 | 0 | 1 | 0 | 0 | 0 | 0 | STATE_0 | 0 | 0 | 0 |
| | | | | 0 | 0 | 1 | 1 | 0 | 0 | 0 | STATE_0 | 0 | 0 | 0 |
| | | | | 0 | 1 | 0 | 0 | 0 | 0 | 0 | STATE_0 | 0 | 0 | 0 |
| | | | | 0 | 1 | 0 | 1 | 0 | 0 | 0 | STATE_0 | 0 | 0 | 0 |
| | | | | 0 | 1 | 1 | 0 | 0 | 0 | 0 | STATE_0 | 0 | 0 | 0 |
| | | | | 0 | 1 | 1 | 1 | 0 | 0 | 0 | STATE_0 | 0 | 0 | 0 |
| | | | | 1 | 0 | 0 | 0 | 0 | 0 | 0 | STATE_1 | 0 | 0 | 1 |
| | | | | 1 | 0 | 0 | 1 | 0 | 0 | 0 | STATE_1 | 0 | 0 | 1 |
| | | | | 1 | 0 | 1 | 0 | 0 | 0 | 0 | STATE_1 | 0 | 0 | 1 |
| | | | | 1 | 0 | 1 | 1 | 0 | 0 | 0 | STATE_1 | 0 | 0 | 1 |
| | | | | 1 | 1 | 0 | 0 | 0 | 0 | 0 | STATE_1 | 0 | 0 | 1 |
| | | | | 1 | 1 | 0 | 1 | 0 | 0 | 0 | STATE_1 | 0 | 0 | 1 |
| | | | | 1 | 1 | 1 | 0 | 0 | 0 | 0 | STATE_1 | 0 | 0 | 1 |
| | | | | 1 | 1 | 1 | 1 | 0 | 0 | 0 | STATE_1 | 0 | 0 | 1 |
| | | | | | | | | | | | | | | |
| STATE_1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | STATE_0 | 0 | 0 | 0 |
| | | | | 0 | 0 | 0 | 1 | 1 | 0 | 0 | STATE_0 | 0 | 0 | 0 |
| | | | | 0 | 0 | 1 | 0 | 1 | 0 | 0 | STATE_0 | 0 | 0 | 0 |
| | | | | 0 | 0 | 1 | 1 | 1 | 0 | 0 | STATE_0 | 0 | 0 | 0 |
| | | | | 0 | 1 | 0 | 0 | 1 | 0 | 0 | STATE_0 | 0 | 0 | 0 |
| | | | | 0 | 1 | 0 | 1 | 1 | 0 | 0 | STATE_0 | 0 | 0 | 0 |
| | | | | 0 | 1 | 1 | 0 | 1 | 0 | 0 | STATE_0 | 0 | 0 | 0 |
| | | | | 0 | 1 | 1 | 1 | 1 | 0 | 0 | STATE_0 | 0 | 0 | 0 |
| | | | | 1 | 0 | 0 | 0 | 1 | 0 | 0 | STATE_1 | 0 | 0 | 1 |
| | | | | 1 | 0 | 0 | 1 | 1 | 0 | 0 | STATE_1 | 0 | 0 | 1 |
| | | | | 1 | 0 | 1 | 0 | 1 | 0 | 0 | STATE_1 | 0 | 0 | 1 |
| | | | | 1 | 0 | 1 | 1 | 1 | 0 | 0 | STATE_1 | 0 | 0 | 1 |
| | | | | 1 | 1 | 0 | 0 | 1 | 0 | 0 | STATE_2 | 0 | 1 | 0 |
| | | | | 1 | 1 | 0 | 1 | 1 | 0 | 0 | STATE_2 | 0 | 1 | 0 |
| | | | | 1 | 1 | 1 | 0 | 1 | 0 | 0 | STATE_2 | 0 | 1 | 0 |
| | | | | 1 | 1 | 1 | 1 | 1 | 0 | 0 | STATE_2 | 0 | 1 | 0 |
| | | | | | | | | | | | | | | |
| STATE_2 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | STATE_0 | 0 | 0 | 0 |
| | | | | 0 | 0 | 0 | 1 | 1 | 1 | 0 | STATE_0 | 0 | 0 | 0 |
| | | | | 0 | 0 | 1 | 0 | 1 | 1 | 0 | STATE_0 | 0 | 0 | 0 |
| | | | | 0 | 0 | 1 | 1 | 1 | 1 | 0 | STATE_0 | 0 | 0 | 0 |
| | | | | 0 | 1 | 0 | 0 | 1 | 1 | 0 | STATE_0 | 0 | 0 | 0 |
| | | | | 0 | 1 | 0 | 1 | 1 | 1 | 0 | STATE_0 | 0 | 0 | 0 |
| | | | | 0 | 1 | 1 | 0 | 1 | 1 | 0 | STATE_0 | 0 | 0 | 0 |
| | | | | 0 | 1 | 1 | 1 | 1 | 1 | 0 | STATE_0 | 0 | 0 | 0 |
| | | | | 1 | 0 | 0 | 0 | 1 | 1 | 0 | STATE_2 | 0 | 1 | 0 |
| | | | | 1 | 0 | 0 | 1 | 1 | 1 | 0 | STATE_2 | 0 | 1 | 0 |
| | | | | 1 | 0 | 1 | 0 | 1 | 1 | 0 | STATE_3 | 0 | 1 | 1 |
| | | | | 1 | 0 | 1 | 1 | 1 | 1 | 0 | STATE_3 | 0 | 1 | 1 |
| | | | | 1 | 1 | 0 | 0 | 1 | 1 | 0 | STATE_2 | 0 | 1 | 0 |
| | | | | 1 | 1 | 0 | 1 | 1 | 1 | 0 | STATE_2 | 0 | 1 | 0 |
| | | | | 1 | 1 | 1 | 0 | 1 | 1 | 0 | STATE_3 | 0 | 1 | 1 |
| | | | | 1 | 1 | 1 | 1 | 1 | 1 | 0 | STATE_3 | 0 | 1 | 1 |

| STATE | STATE_BITS | | | INPUTS | | | | OUTPUTS | | | NEXT_STATE | NEXT_STATE_BITS | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | input_0 | input_1 | input_2 | input_3 | output_0 | output_1 | output_2 | | | | |
| STATE_3 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | STATE_0 | 0 | 0 | 0 |
| | | | | 0 | 0 | 0 | 1 | 1 | 1 | 0 | STATE_0 | 0 | 0 | 0 |
| | | | | 0 | 0 | 1 | 0 | 1 | 1 | 0 | STATE_0 | 0 | 0 | 0 |
| | | | | 0 | 0 | 1 | 1 | 1 | 1 | 0 | STATE_0 | 0 | 0 | 0 |
| | | | | 0 | 1 | 0 | 0 | 1 | 1 | 0 | STATE_0 | 0 | 0 | 0 |
| | | | | 0 | 1 | 0 | 1 | 1 | 1 | 0 | STATE_0 | 0 | 0 | 0 |
| | | | | 0 | 1 | 1 | 0 | 1 | 1 | 0 | STATE_0 | 0 | 0 | 0 |
| | | | | 0 | 1 | 1 | 1 | 1 | 1 | 0 | STATE_0 | 0 | 0 | 0 |
| | | | | 1 | 0 | 0 | 0 | 1 | 1 | 0 | STATE_2 | 0 | 1 | 0 |
| | | | | 1 | 0 | 0 | 1 | 1 | 1 | 0 | STATE_2 | 0 | 1 | 0 |
| | | | | 1 | 0 | 1 | 0 | 1 | 1 | 0 | STATE_3 | 0 | 1 | 1 |
| | | | | 1 | 0 | 1 | 1 | 1 | 1 | 0 | STATE_4 | 1 | 0 | 0 |
| | | | | 1 | 1 | 0 | 0 | 1 | 1 | 0 | STATE_2 | 0 | 1 | 0 |
| | | | | 1 | 1 | 0 | 1 | 1 | 1 | 0 | STATE_2 | 0 | 1 | 0 |
| | | | | 1 | 1 | 1 | 0 | 1 | 1 | 0 | STATE_3 | 0 | 1 | 1 |
| | | | | 1 | 1 | 1 | 1 | 1 | 1 | 0 | STATE_4 | 1 | 0 | 0 |
| | | | | | | | | | | | | | | |
| STATE_4 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | STATE_0 | 0 | 0 | 0 |
| | | | | 0 | 0 | 0 | 1 | 1 | 1 | 1 | STATE_0 | 0 | 0 | 0 |
| | | | | 0 | 0 | 1 | 0 | 1 | 1 | 1 | STATE_0 | 0 | 0 | 0 |
| | | | | 0 | 0 | 1 | 1 | 1 | 1 | 1 | STATE_0 | 0 | 0 | 0 |
| | | | | 0 | 1 | 0 | 0 | 1 | 1 | 1 | STATE_0 | 0 | 0 | 0 |
| | | | | 0 | 1 | 0 | 1 | 1 | 1 | 1 | STATE_0 | 0 | 0 | 0 |
| | | | | 0 | 1 | 1 | 0 | 1 | 1 | 1 | STATE_0 | 0 | 0 | 0 |
| | | | | 0 | 1 | 1 | 1 | 1 | 1 | 1 | STATE_0 | 0 | 0 | 0 |
| | | | | 1 | 0 | 0 | 0 | 1 | 1 | 1 | STATE_2 | 0 | 1 | 0 |
| | | | | 1 | 0 | 0 | 1 | 1 | 1 | 1 | STATE_2 | 0 | 1 | 0 |
| | | | | 1 | 0 | 1 | 0 | 1 | 1 | 1 | STATE_4 | 1 | 0 | 0 |
| | | | | 1 | 0 | 1 | 1 | 1 | 1 | 1 | STATE_4 | 1 | 0 | 0 |
| | | | | 1 | 1 | 0 | 0 | 1 | 1 | 1 | STATE_2 | 0 | 1 | 0 |
| | | | | 1 | 1 | 0 | 1 | 1 | 1 | 1 | STATE_2 | 0 | 1 | 0 |
| | | | | 1 | 1 | 1 | 0 | 1 | 1 | 1 | STATE_2 | 0 | 1 | 0 |
| | | | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | STATE_2 | 0 | 1 | 0 |

# C Python txt-file parser script

```
#Parameters describing the instantiated RTL FSM
num_state_bits  = 3
num_output_bits = 3
num_input_bits  = 4

#Parameters describing the txt file with FSM program
#If the RTL has more i/o and state than the state table
num_state_bits_in_file = 2
num_output_bits_in_file = 2
num_input_bits_in_file = 2

STD = open("state_transition_table.txt","r") #Specify txt file here

#Loop through the .txt line by line
for line in STD:
  temp_state = ""
  temp_output = ""
  temp = ""

  #Append zeros for unused state bits
  for i in range(0,num_state_bits - num_state_bits_in_file):
    temp_state = temp_state + "0"

  #Append zeros for unused output bits
  for i in range(0,num_output_bits - num_output_bits_in_file):
    temp_output = temp_output + "0"

  #Add the actual next state and output bits to the zeros
  for i in range(0,num_state_bits_in_file):
    temp_state = temp_state + line[i]
  for i in range(num_state_bits_in_file, num_state_bits_in_file + num_output_bits_in_file):
    temp_output = temp_output + line[i]

  temp = temp_state + temp_output

  #Create bus driver code
  print("ucl_BusDriver.ta_simpleWrite (.addr (ID_PROGRAMMABLESTATEMACHINE_RW_BASE), .data (32'b"+temp+"));")


```



NTNU
Norwegian University of
Science and Technology