NTNU – Trondheim
Norwegian University of
Science and Technology

Self-cloning state machines on FPGA

Daniel H Blomkvist

Master of Science in Electronics
Submission date: June 2013
Supervisor: Kjetil Svarstad, IET

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Project description

It is the purpose to design a state representation on FPGA that can be used to construct State Machines that should be able to clone themselves. This is an interesting operation for representation of non-determinism in run time reconfiguration.

Abstract

The purpose of this thesis is to continue development of a single state representation for use in a Self-Cloning State Machine. The Self-Cloning State Machine can be used to represent Non-Deterministic Finite State Machines on a FPGA. NFSMs can be used to for instance match regular expressions and the Self-Cloning State Machine is a run-time reconfiguration based approach to this problem. Previous work has been performed on design tool experiments, choosing a FPGA vendor, and creating a definition of the singular state representation. This continuation focuses on further development of the singular state and creating a system around it for experimentation.

A system is designed to be used as an example model of a Self-Cloning State Machine for the Virtex-4 FPGA. This system consists of several finite-state machines built up by several singular states, a control system, and a result management module. The FSMs are connected together in order to model a previously defined NFSM. To model the NFSM a control system is made to enable and disable the different FSMs as the transitions happen in the NFSM. Each of these FSMs have their own data path used for computing a simple operation based on a Multiply-Accumulate Circuit. The data path uses one-hot coded state vectors from the FSM as its internal state machine. A deterministic version is also created to be able to compare the area and functional differences. A comparison between the Self-Cloning State Machine and another method of implementing a NFSM is made. There is also a brief comparison of two different implementation methods for run-time reconfiguration with respect to ease of implementation. The two methods are a framework for run-time reconfiguration and a Virtual FPGA system.

The modifications made to the singular state representation are found to be appropriate for allowing the singular state to represent any state in any FSM. A problem was discovered with the size of the configuration register but the problem can be solved. The control system, data path and result management system functioned correctly and they were well suited to show the functionality of a system like this one. Comparing the Self-Cloning State Machine to another implementation method for NFSMs shows that this implementation demands much more physical area which is a disadvantage, but the Self-Cloning State Machine may be more flexible than the other method. The V-FPGA method of performing run-time reconfiguration is found to be superior to another type of framework created specifically for the Virtex-4 FPGA.

Sammendrag

Hensikten med denne oppgaven er aa viderefoere utviklingen av en enkelttilstandsrepresentasjon til bruk i en selv-klonende tilstandsmaskin. En selvklonende tilstandsmaskin kan brukes til aa representere ikke-deterministiske tilstandsmaskiner (NFSMs) paa en FPGA. NFSMs kan brukes til for eksempel matche regular expressions og selvklonende tilstandsmaskiner er en kjoeretidsrekonfigureringstilnaerming til dette problemet. Tidligere arbeid har blitt utfoert innenfor designverktoeyeksperimenter, det aa velge en FPGA leverandoer, og det aa skape en definisjon av enkelttilstandsrepresentasjon. Denne fortsettelsen fokuserer paa videreutvikling av enkelttilstanden og det aa lage et system rundt den for eksperimentering.

Et system er konstruert for aa bli brukt som et eksempel paa modell av en selvklonende tilstandsmaskin for en Virtex-4 FPGA. Dette systemet bestaar av flere tilstandsmaskiner bygget opp av flere enkelttilstander, et kontrollsystem, og en modul for resultatbehandling. De tilstandsmaskinene er koblet sammen for aa modellere en tidligere definert NFSM. For aa modellere en NFSM er et kontrollsystem laget for aa aktivere og deaktivere de forskjellige tilstandsmaskinene ettersom overgangene skjer i NFSM. Hver av disse tilstandsmaskinene har sine egne dataveier som brukes for aa beregne en enkel operasjon basert paa en multiply-accumulatekrets. Dataveien bruker one-hot kodede tilstandsvektorer fra en tilstandsmaskin som deres interne tilstandsmaskin. En deterministisk versjon er ogsaa laget for aa kunne sammenligne areal og funksjonelle forskjeller. En sammenligning mellom den selvklonende tilstandsmaskinen og en annen metode for aa implementere en NFSM er gjort. Det er ogsaa gjort en kort sammenligning av to forskjellige metoder for aa gjennomfoering kjoeretidsrekonfigurering med hensyn til enkel implementasjon. De to metodene er et rammeverk for kjoeretidsrekonfigurering og et system for design av virtuelle FPGAer.

Endringene som er gjort i enkelttilstandsrepresentasjonen er funnet aa vaere hensiktsmessig for aa la enkelttilstanden representere hvilken som helst tilstand i hvilken som helst tilstandsmaksin. Et problem ble paavist med stoerrelsen paa konfigurasjonensregisteret, men problemet kan loeses. Kontrollsystemet, databanen og resultatstyringssystemet fungerte korrekt og de var godt egnet til aa vise funksjonaliteten til et system som dette. Sammenligningen av en selvklonende tilstandsmaskin med en annen implementeringsmetode for NFSMer viser at denne implementeringen krever mye mer areal, noe som er en ulempe, men selvklonende tilstandsmaskiner kan vaere mer fleksibel enn den andre metoden. V-FPGA-metoden for aa utfoere kjoeretidsrekonfigurasjon er funnet aa vaere overlegen i forhold til en annen type rammeverk laget spesielt for en Virtex-4 FPGA.

Preface

Thanks to my advisor Professor Kjetil Svarstad for providing this task and guiding me through it.

Contents

List of Figures

List of Tables

Abbreviations

DFSM	**D**eterministic **F**inite **S**tate **M**achine
FPGA	**F**ield-**P**rogrammable **G**ate **A**rray
FSM	**F**inite **S**tate **M**achine
HDL	**H**ardware **D**escription **L**anguage
HMP	**H**ard **M**acro **P**orts
IOB	**I**nput **O**utput **B**uffer
MAC	**M**ultiply-**A**ccumulate **C**ircuit
MMAC	**M**odified **M**ultiply-**A**ccumulate **C**ircuit
NFSM	**N**on-deterministic **F**inite **S**tate **M**achine
V-FPGA	**V**irtual **F**ield-**P**rogrammable **G**ate **A**rray

Chapter 1

Introduction

1.1 Problem Clarification

Previously a singular state representation for a self-cloning state machine has been investigated in a project [Blomkvist, 2012]. This thesis is a continuation of that work. The point of this thesis is to further investigate whether or not a self-cloning state machine is a viable solution for representing Non-Deterministic Finite State Machines on FPGAs. While the project was very detail oriented with respect to the implementation methods this thesis will try to have a broader perspective and address system-relevant issues related to this subject. Seeing as this is a very large subject area, and very little work has been done before, small incremental steps are taken in the design. The scope of this work is:

- To make a system level analysis to determine system requirements for control and result management.
- To implement a model of a NFSM on an FPGA in VHDL and compare this to a straight-forward DFSM implementation with a few different metrics, e.g. speed and area.
- To review different implementation methods for a true NFSM based on the implementation examples.

The run-time reconfiguration itself will not be implemented here, different possible methods will simply be discussed with regards to the results from the implementation of the model.

1.2 Motivation

Traditionally there has always been a clear distinction between the hardware approach of the application-specific integrated circuit and the software approach of the microprocessor. With the rise of the field-programmable gate array the line between hardware and software is becoming more and more blurry. The reconfigurability of the FPGA allows bug-fixing and shorter time-to-customer than ASICs, while still being faster than the microprocessor. Many applications use FPGA modules as hardware accelerators for microprocessors and many FPGAs include microprocessors embedded on-chip. Since the FPGA can offer some of the flexibility of software and the speed and parallel computing of hardware it helps bridge the gap between hardware and software.

Run-time reconfiguration of FPGAs can further bridge the hardware/software gap by enabling the FPGA to change functionality in run-time and react to the run-time conditions of the system. This can bring very high levels of flexibility to a system but the design complexity is severe at the moment. To simplify run-time reconfiguration new methods can be developed. One such method is the Self-Cloning State machine. Provided that a good work flow can be developed the Self-Cloning State machine can turn out to be an easier way to perform run-time reconfiguration which is device independent and excellent for use with any application that requires an unknown pattern to be matched.

The motivation for this work is finding out if self-cloning state machines are a viable alternative for designing run-time reconfigurable systems. Not all applications will be suited for this kind of system but some applications which require patterns to be recognized (e.g. regular expression matching) could possibly produce faster results by using NFSMs in general ([Sidhu and Prasanna, 2001])and self-cloning state machines in particular.

The contribution to this subject area by this thesis is a redefinition and re-implementation of a singular state representation for use in a Self-Cloning State Machine, a framework for testing this state beginning with a systems presentation and analysis, and comparing this with the same concept implemented in a normal DFSM. Further it moves on to a brief comparison with another method of implementing Non-deterministic State Machines and another brief analysis of what kind of reconfiguration solution to choose for the Self-Cloning State Machine.

1.3 Report structure

The thesis is structured as follows. Chapter two gives some background information about what this thesis is built on. Chapter three is the theory chapter

where the field-programmable gate array, run-time reconfiguration, the non-deterministic state machine, and virtual hardware is explained. Chapter four analyses and defines the demands and limitations of the self-cloning state machine model created here. Chapter five is the chapter where implementation examples are explained. A modified multiply-accumulate operation is implemented in both a normal straight-forward deterministic finite-state machine and in a model of a self-cloning non-deterministic finite state machine. Chapter six is the results chapter. Chapter seven is a discussion of the results. Chapter nine is the conclusion.

Chapter 2

Background

2.1 Compiling Regular Expressions into Non-Deterministic State Machines for Simulation in SystemC

This Master's Thesis written by Kjetil Volden in 2010 [Volden, 2011] models regular expression matching on FPGAs using SystemC by first defining a specific regular expression syntax for that purpose, and then creating a system for compiling regular expressions into non-deterministic finite state machines. Since SystemC does not allow for creation and deletion of modules during run-time he chose to model it by activating and deactivating modules depending on what regular expression was sent into the system. The thesis ends by suggesting further work on trying to implement non-deterministic finite state machines on FPGA.

2.2 Self-Cloning State Machines on FPGA specialization project

The project report [Blomkvist, 2012] consisted of an initial survey into which vendor would be the best suited for this application. Following that were experiments regarding what kind of synthesis parameters would be the best for an initial singular state representation, i.e., whether to choose one-hot or Gray coding for the internal state machine of the singular state and whether to optimize for area or speed. Manually designing a state machine by hand in a Xilinx tool called FPGA Editor was performed to learn more about both the FPGA

Editor tool and the architecture of the Xilinx Virtex-series FPGA. Finally a preliminary definition of a singular state was made. A limited investigation into how the copy operation could be performed was also made. The projects concludes that using Xilinx as the vendor is the best choice since Xilinx supports run-time reconfiguration, one-hot coding and optimizing for area is the best starting point when designing the singular state, and that the singular state needs more work to make it generic.

2.3 Fast Regular Expression Matching using FPGAs

Some work has been carried out previously on implementing a Non-deterministic Finite State Machine on an FPGA [Sidhu and Prasanna, 2001]. According to the article this was the first practical use of a NFSM on programmable logic. An algorithm was made that quickly constructs a NFSM in run-time based on a regular expression which is to be matched by input data. Each state in the NFSM is set to be one flip-flop using the one-hot encoding scheme. The non-deterministic transitions are made by simply connecting the output from a source state to the input of all the destination states. ϵ-moves are made by connecting the input signal of a source state to the input signal of the destination state, allowing a transition to the destination state without waiting for the symbol to be processed. The NFSM is constructed using predefined building blocks representing the different operators of the regular expression syntax. Their system showed good performance compared with serial regular expression matching algorithms.

Figure 2.1 shows the regular expression modules of this approach. The single character module in (a) takes in a text character and a value from the previous module. If the character is present in the regular expression the character register will be set high, and if the i signal is also high because the NFSM is in the previous state this state will be set high. This can be seen in (c) where N1 has to come before N2, so N1 will be checked first, and will be set high before N2 is checked. The structure in (b) shows how two single character modules are connected to form an OR-operation in the regular expression by connecting the i signal to the inputs of both single characters and the outputs to an OR-gate. The last structure is the one in (d) where zero or more of the same character is accepted as a valid string because of the connections with the OR-gates. The initial input to the i signal is always set high to allow the circuit to function.

This is relevant as a comparison since it gives an alternative approach to designing NFSMs and can be a comparison to using the Self-Cloning State Machine, especially with regards to area usage.

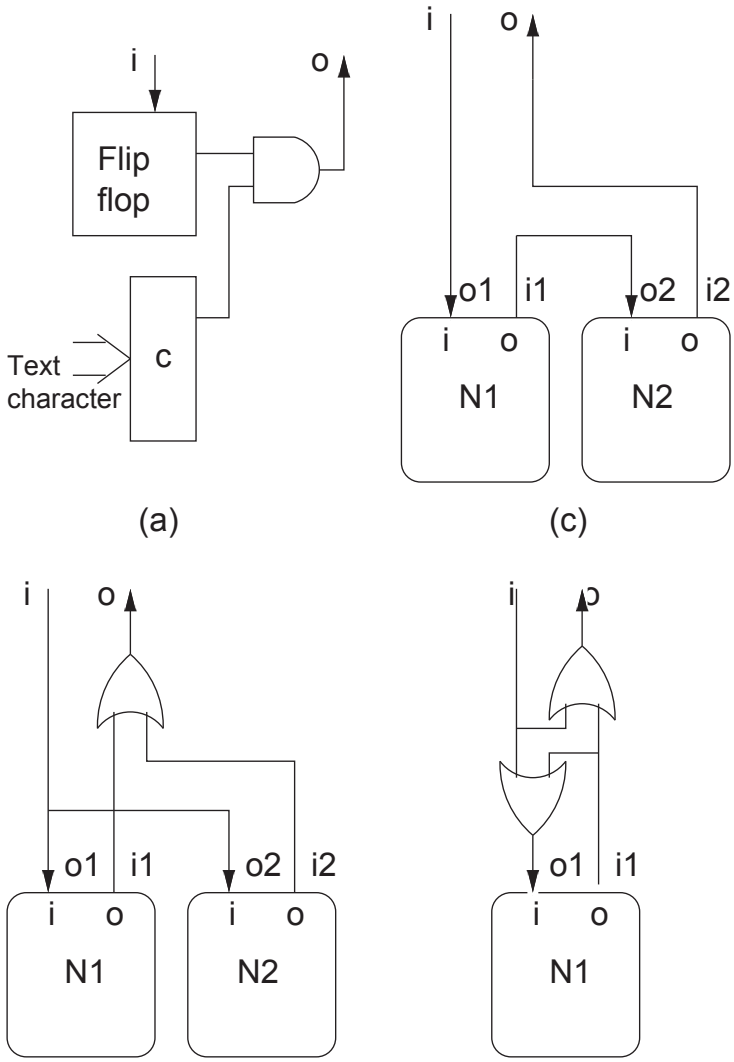

FIGURE 2.1: Logic structures for (a) single character, (b)$r_1|r_2$, (c) $r_1 \cdot r_2$, (d) r_1* [Sidhu and Prasanna, 2001]

Chapter 3

Theory

3.1 Field-Programmable Gate Arrays

The Field-Programmable Gate Array (FPGA) is an electronic circuit that can be reconfigured several times [Compton and Hauck, 2002]. Usually the configuration memory of an FPGA is volatile so that it needs to be reconfigured every time it's powered on, but other types of memory exist. The advantage of the volatile type of memory is that it is faster. Figure 3.1 shows the architecture of interconnect and logic cells in a typical FPGA. The Logic Blocks are the part of the FPGA doing the actual computations and are built up of look-up tables (LUTs), flip-flops, multiplexers and sometimes carry chains. The definition of what a logic block is varies between different FPGA vendors but the smallest logic cell (figure 3.2) usually consists of a LUT, a D-type flip-flop and a 2-to-1 multiplexer used for bypassing the flip-flop when needed. The LUTs are 1-bit asynchronous RAMs [Ashenden, 2008] which can be configured to do a high number of operations. All input combinations are passed through AND-gates and then the programmable memory cell chooses which of the signals from the AND-gates to accept and sets the LUT to be high when that combination is active on the inputs [Bailey, 2011]. The number of operations a LUT can perform depends on the number of inputs. The equation $n = 2^{2^k}$ describes the relationship between the k number of inputs and n number of possible operations. So for a 3-input LUT the number of operations possible is 256, for a 4-input LUT the number is 65536 and so forth. This means that the area required for implementing a LUT increases more than exponentially with the number of inputs. The other types of blocks are connect blocks and switching blocks. Connect blocks is the interface between the global routing network and the logic blocks. The switching blocks are used for changing routing direction between horizontal and vertical lines.

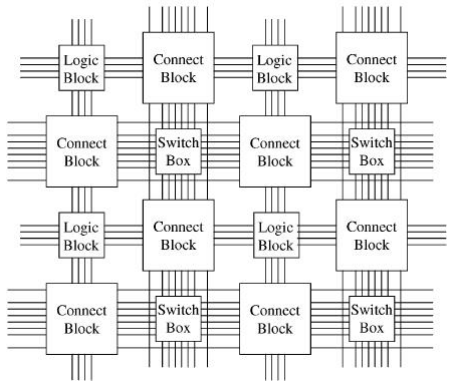

FIGURE 3.1: FPGA top level architecture schematic [Compton and Hauck, 2002]

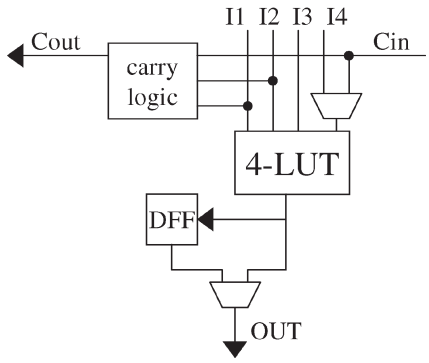

FIGURE 3.2: A typical logic cell [Compton and Hauck, 2002]

FPGAs offer advantages that other types of electronic circuits do not. Application-Specific Integrated Circuits are fast and use low amounts of energy, but they are inflexible since they can't be changed after production. In addition they are very expensive to manufacture and thus an FPGA can often be a better choice in low-volume production. Microprocessors are flexible but lack the speed and often the parallel computing capabilities of an FPGA. The FPGA can be changed after the product has been shipped to a customer if an error is discovered, and many FPGA vendors provide a microprocessor for the FPGA either as a soft core processor or actual embedded hardware embedded in the FPGA. In that way the FPGA can provide both the serial processing of the microprocessor and the parallel processing of the FPGA structure. Fully utilizing this requires that developers know how to partition the system into software and hardware components.

Designing for an FPGA usually requires that the designer describes the functionality in a hardware description language (HDL) and then takes advantage of the tools offered by the FPGA manufacturer used in the design. The tools are usually specific for that particular vendor, and sometimes for a particular device family.

3.2 Run-Time Reconfiguration

An FPGA is usually configured before it is activated in a system, however, in certain applications it can be useful to change the configuration of an FPGA when it's running. Run-time reconfiguration allows this to happen [Compton and Hauck, 2002]. In order to change the functionality of the FPGA in run-time the operation has to be halted before swapping the old configuration with the new one. Useful applications for run-time reconfiguration are any applications where optimizations can be done in run-time, new and updated standards are frequent, or other applications where there is a high chance of needing something to be changed [Koch et al., 2012]. Run-time reconfiguration is closely related to the concept of virtual hardware explained in the next section.

There are a few different types of run-time reconfiguration. Single-context devices need to change the whole configuration serially every time a context-switch needs to happen. This takes several milliseconds to complete and should happen very rarely. Multi-context devices can save multiple contexts on the FPGA to make the context switch happen in several orders of magnitude faster. Partially reconfigurable devices is able to halt only parts of the FPGA while letting the rest of the FPGA run as normal in order to change the configuration. Addressing the correct configuration bits in the configuration memory lets the system change only the exact parts that need to be changed. When using partial reconfiguration the FPGA is divided into a static part which should never be changed, and a dynamic part which can be changed in run-time.

Self-reconfiguration is a subcategory of run-time reconfiguration where the FPGA does the reconfiguration itself. Some FPGAs support this by allowing on-chip logic to connect to the configuration logic. This can be useful where it is not practical to organize the run-time reconfiguration from the outside world because of overhead or space limitations. The control logic needs to be in the static part since it must remain unchanged and has to be able to keep running at all times.

3.3 Virtual FPGA Architectures

Hardware virtualization means that an application executes on virtual hardware as opposed to physical hardware [Plessl and Platzner, 2004]. That means that

an abstraction layer is put on top of the physical hardware to separate the design process so that the designer does not need to worry about the physical structure of the hardware. Virtual design units are made up of several hardware design units (i.e. CLBs) to create the abstraction like in figure 3.3.

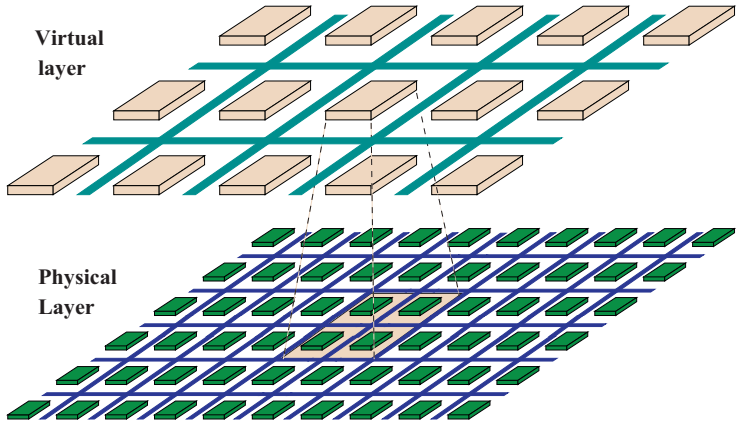

FIGURE 3.3: Physical and virtual layers [Lagadec et al., 2001]

Virtual hardware is analogous to virtual memory which is used to allow applications to address a larger memory than physically exists. Similarly a system using virtual hardware can swap configurations in and out of the physical hardware in run-time allowing the system to use a larger amount of hardware than physically exists. Using virtual hardware increased independence between the mapped architectures and the capacity of the target architecture can be obtained in exchange for speed. Lower speed can however be remedied by specializing the virtual layer towards a particular application by making fast specific virtual hardware for that application [Lagadec et al., 2001].

There are several different approaches to virtualization of hardware. Temporal partitioning splits the application into several smaller parts which fit on the physical architecture and execute the parts sequentially. Virtualized execution is another method. The goal of Virtualized execution is to gain a large level of independence within a device family. The application is divided into atomic units of computation. The whole system is then described by these tasks and the interactions between them. All the devices in the family support the task and interaction definitions, so all the devices in the family can implement these

abstractions. The difference is how many tasks can run concurrently depending on the size of the FPGA the application is implemented on. Task scheduling is one of the challenges using this method. The third option is the Virtual Machine method. A higher level of device independence is provided by the virtual machine method since the application is mapped to abstract computing architecture instead of being mapped for a particular family. This can be done in two ways. Either remap the application to the native application code of the target architecture, or the hardware architecture can run an interpreter that executes the abstract application code.

A way of implementing a virtual FPGA architecture is by coupling an FPGA with a microprocessor [Hubner et al., 2011]. This has previously been implemented in order to use virtual FPGAs as hardware accelerators serving a microprocessor. Figure 3.4 shows the architecture of the whole system. The system is built up of the microprocessor, one or more virtual FPGA cores, a configuration controller, en external memory used for storing configuration and other data, and an AMBA bus ([ARM, n.d.]) used for on-chip communication. The processor used is an ARM Cortex-M1 inside the FPGA itself and is used for serial execution including control and interfacing. Memory read and write is done through the AMBA High-speed Bus (AHB) which both the processor and the configuration controller are masters of. The processor communicates with both the configuration controller and the V-FPGA cores through the AMBA Peripheral Bus (APB). The processor tells the configuration controller which configuration to load into which V-FPGA via the APB. So the configuration controller is a slave of the APB but a master of the AHB. The V-FPGA cores are slaves of both the APB and the configuration bus. The configuration bus out of the configuration controller is used in combination with chip select signals to control which of the V-FPGAs are configured with the data on the bus. That means that any number of V-FPGAs can be configured at the same time with the same configuration enabling dynamic and partial reconfiguration. The architecture of the V-FPGAs themselves uses a uniform 2-dimensional grid structure with Programmable Switch Matrices (PSMs) and Complex Logic Blocks (CLBs) with I/O Blocks (IOBs) at the borders. A CLB consists of a 4-input LUT, a D-type flip-flop, a configuration register and some other logic. The PSMs are used for routing between the CLBs, and the IOBs are used to connect to the outside world. A custom tool-chain is needed for this type of implementation because none of the vendors support it natively.

3.4 The Non-deterministic Finite State Machine

Automata theory is the theory about the properties of problems that are solvable by computers [Harvey, 1997]. The purpose of automata theory is to provide a general model of computing that is independent of the implementation method. One such model is the Finite State Machine (FSM). The finite state machine

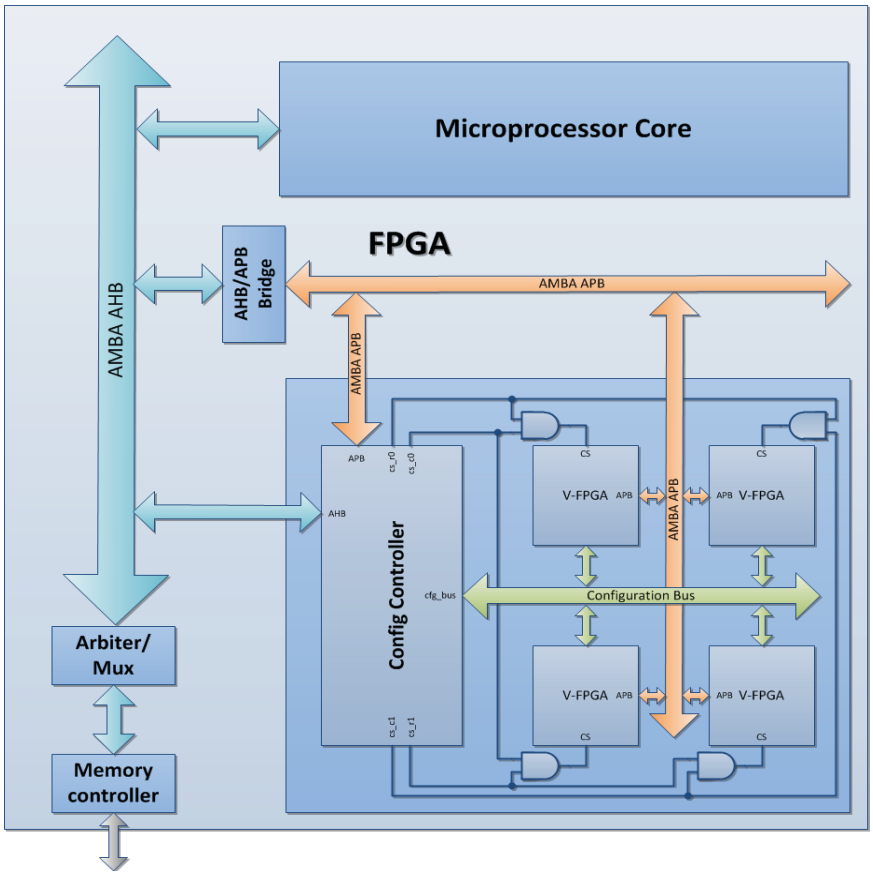

FIGURE 3.4: Overview of the virtualization system architecture [Hubner et al., 2011]

is often described deterministically, i.e. given a known current state and/or a known input the next state in the computation is also known because all of the state transitions are given. This type of finite state machine is called a Deterministic Finite State Machine (DFSM).

The Non-deterministic Finite State Machine (NFSM), however, is not uniquely defined for each state and each input. The NFSM is a state machine which can have several state transitions from the same state triggered by the same input going to different destination states. The NFSM cannot solve any more complex problems than an ordinary DFSM, it can in fact be converted to a DFSM by use of powerset construction, but it can be more practical in use for some applications. An example is a string matcher like the one in figure 3.5. The string in the figure should start with an "A" and end with a "C" but any

number of "A", "B" or "C" can be in between the first "A" and the last "C". When the first "A" has been received the NFSM will transition from the first state to the second and then it will be able to receive all symbols indefinitely. As long as an "A" or a "B" is received the machine will stay in the second state, but if a "C" is received it can either transition further to the last state or stay in the second state. Since the string can end with a "C", but not necessarily does, the machine can end up in both states and the outside world cannot know which one it ended up in. A way to solve this is to copy the state machine every time a "C" arrives and then let one of the machines be in the second state and one be in the third state. That way the string will be detected but the machine will still keep looking for more strings.

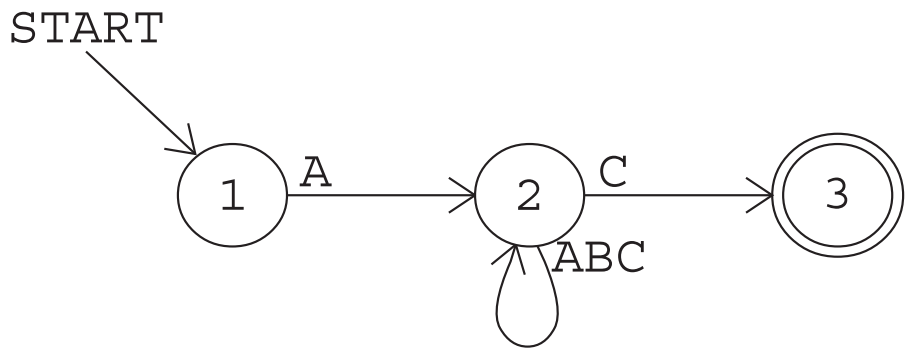

FIGURE 3.5: String matcher in a NFSM [Harvey, 1997]

Figure 3.6 shows how it would be done in a DFSM. The difference is that when a "C" arrives the machine transitions to the thirds state regardless. Here the third state is an accept state, but the machine can and will transition away from the third state when a new character arrives. That means that the machine isn't done even if it ends in the accept state. Both types need to accept sub-strings in run-time since they can't know if the last "C" has arrived, but the NFSM has the advantage of fewer transitions and better separation between the different stages of the machine since it's done when it ends up in the third state. In the NFSM there is a clear distinction between the first character, the middle characters and the last character, while in the DFSM there is a larger number of transitions and there is no separation between the second and last stages of the string.

The definition of a NFSM is given by the following five-tuple [Svarstad and Volden, 2011]:

- Q is the *set of states*
- Σ is the finite *alphabet*
- q_0 is the *initial state*, $q_0 \in Q$

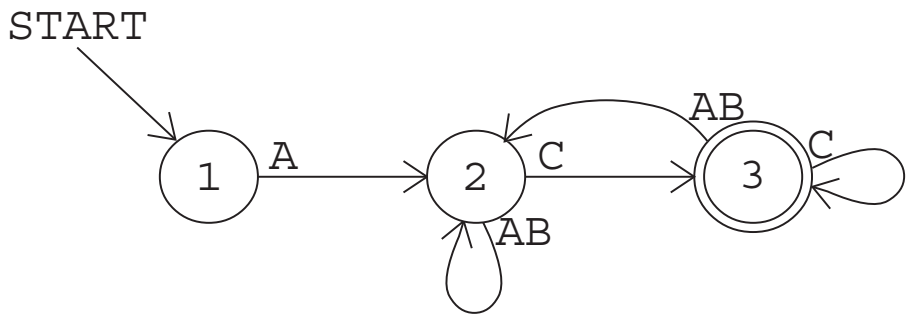

FIGURE 3.6: String matcher in a DFSM [Harvey, 1997]

- A is the *set of final states*, $A \subseteq Q$
- δ is the state transition relation: $\delta : Q \times \Sigma \rightarrow 2^Q$

The definition is similar to the DFSM except for the state transition relation which transitions to all subsets (the power set) of Q instead of just Q. The reason is that the NFSM can transition to several states given the same conditions and it is impossible to know which one it is in. The NFSM can also have ϵ-moves, moves which do not require that an input symbol is consumed to transition to the next state [Sidhu and Prasanna, 2001].

Chapter 4

A system level view

Viewing a self-cloning state machine from a system perspective is useful for understanding how it can be of practical use. System perspective means that the implementation details are less important than the overhead system considerations that need to be made. The broader, more abstract concepts are considered to determine some system requirements. To determine these system requirements a circuit based on a multiply-accumulate (MAC) circuit combined with both a deterministic finite-state machine (DFSM) and a model of a non-deterministic finite-state machine (NFSM) will be considered.

The internal state machine of the MAC operation will be controlled by values coming from the detector in [Svarstad and Volden, 2011], which detects three-length sub-strings of ones in a continuous stream of values. This is used as an example because the concept of the detector is already explained in the paper and is simple enough to understand but still complicated enough to model most of what is needed for a system discussion. Figure 4.1 shows the state machine diagram of the substring detector. The only non-deterministic state transition here happens when a "1" is received in the initial state and the transition can either go to S1 again or to S2. Further, if a "1" is received continuously the state machine will eventually end up in S4, the accept state. The way the self-cloning state machine will cope with the non-deterministic transition is to clone itself every time it occurs. Every time a "1" arrives on the input a new clone will be made which explores one of the possibilities, i.e, that the transition went to S2. The other state machine will explore the other possibility, i.e., that the transition went back to S1. This will continue every time a new "1" arrives. For this particular NFSM, however, the maximum number of state machines active at any time will be four. The reason is that the state machines will be pruned when they reach the accept state since the operation then will be done, freeing the area of the FPGA for a new clone. If a "0" arrives on the input all of the currently active clones except one should be pruned and the last clone should

be active in the initial state waiting for another "1" to arrive on the input and the operation to start from the beginning again.

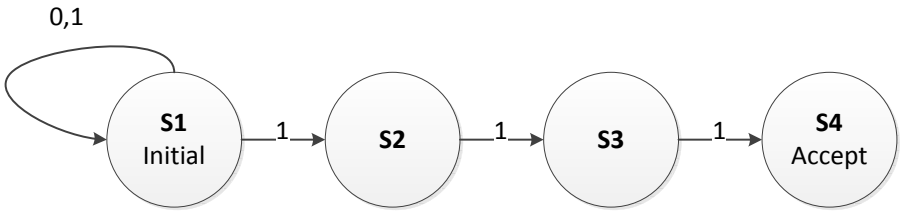

FIGURE 4.1: The modeled NFSM [Svarstad and Volden, 2011]

4.1 Multiply-Accumulate

The MAC operation is quite simple as shown in figure 4.2. The operation consists of first multiplying two numbers and then adding it to an accumulating sum. For this particular application a few changes need to be made to the concept of a MAC circuit. Instead of multiplying two variables and then accumulating it in a register, two variables can be multiplied and then add the result to a third variable. The point of this change is to create a relatively simple data path for use with the non-deterministic state machine previously defined. The modified MAC-circuit (MMAC) has the same number of steps for completion as the NFSM we have chosen to model. The modified circuit is shown in figure 4.3.

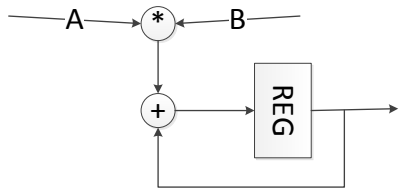

FIGURE 4.2: A MAC circuit

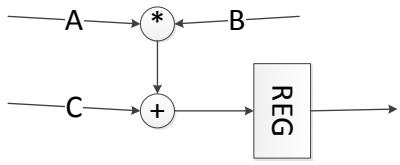

FIGURE 4.3: The modified MAC circuit structure

4.2 Non-deterministic model

The data flow of the MMAC circuit in the NFSM model is shown in figure 4.4. Since there will only exist a maximum of four state machines in this implementation only four are shown in the figure. The input data is accompanied by a high state transition signal to the state machine. Letters are used in the figure only to indicate different data values, for instance different values of an integer, not actual letters. When the "a" arrives the first machine is activated and the "a" is set in the register of the first machine, r^{I}. When the "b" arrives the first machine multiplies it with the previously received "a" and again sends it back to r^{I}. The second machine will just load "b" into r^{II}. This will continue as long as there is input data with a high transition signal coming into the system with results starting to arrive after four clock cycles and then arriving every clock cycle, sort of like a pipeline except none of the machines will be sharing computing resources. When the result is ready the machine can be deactivated as shown by using an "X" in the figure. What will actually happen is that the machine will become active again but start from the beginning as the next machine in the chain, i.e., machine "I" will become machine "V" and so forth.

4.2.1 The singular state

Cloning a state machine requires all of the physical logic in the state machine to be defined and have clear boundaries. The reason is that it simplifies runtime reconfiguration. A control system needs to be able to determine where each of the clones are physically on the FPGA in order to determine if there is room enough for another clone and where on the FPGA there is room. Having exact information about how large a clone is helps with that. A previously defined singular state can be used to create a fully functional state machine which in turn can be used as a NFSM [Blomkvist, 2012]. This singular state can represent one bit in a one-hot coded FSM.

The previously defined singular state was not generic, and work needs to be done to change that. Another problem with the previous definition is that it is very complicated in the sense that it includes a lot of functionality which is not necessarily useful to have in the singular state itself. Redefining it after having built a rudimentary system around it will allow for a definition which is more accurate. Stripping away some things from the previous definition the system demands for the singular state are:

- It should be minimal. That is, it should use as few logic elements as possible.
- It should have clear boundaries for reasons explained previously.
- It should be generic, which means it should be usable as any state in any FSM.

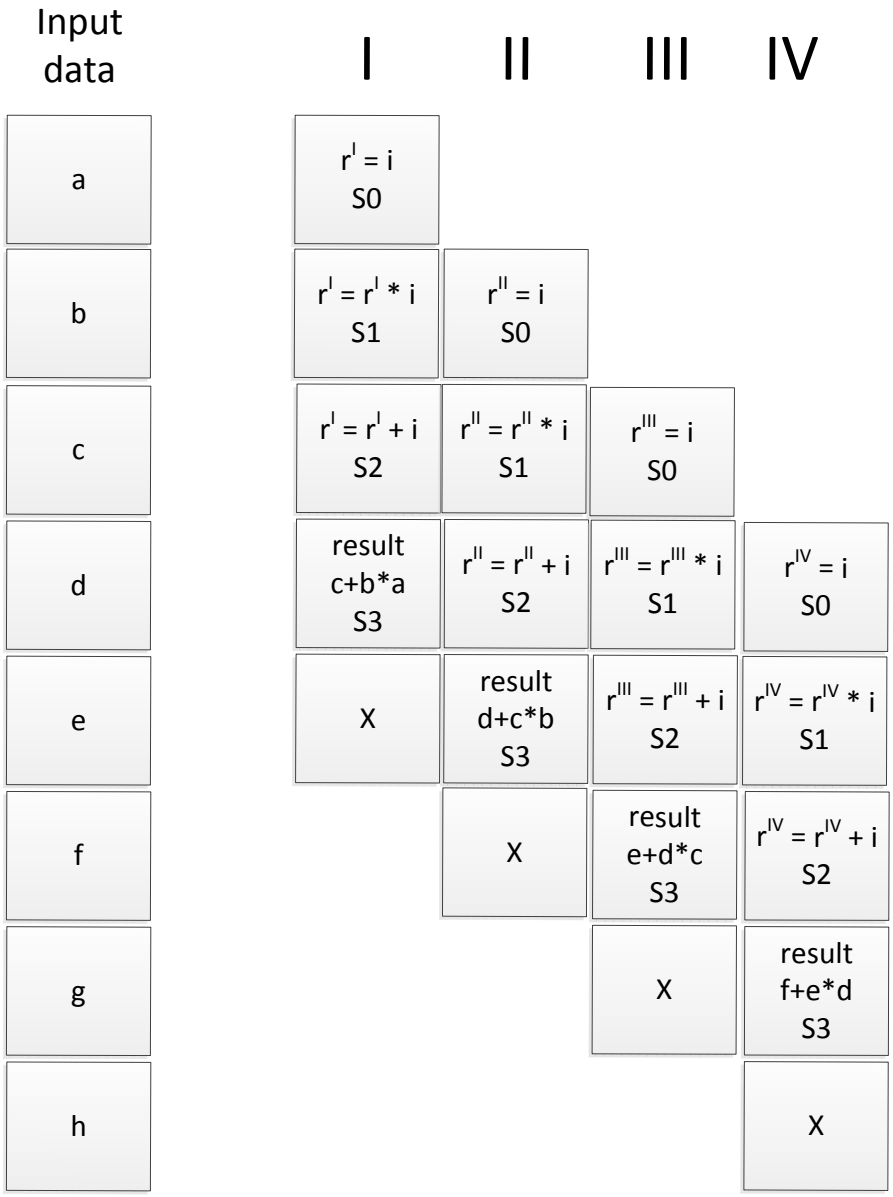

FIGURE 4.4: NFSM MMAC flow

These demands are very concrete and a better starting point than the scattergun approach from the specialization project.

4.2.2 Data path

Any state machine needs to be connected to a data path. Since the data path needs to be connected to the state machine there is an issue with how to deal with the data path when cloning the machine. Since all of the operations in the different state machines happen concurrently each state machine needs its own separate data path. To solve this issue one could determine the maximum number of state machines active in the particular application and then implement that number of data paths in the system. The advantage is having a fixed area used for data path logic only and more predictability regarding how much space is needed. The disadvantage would be that the physical resources will be occupied at all times, even if there aren't enough state machines to use all of the data paths. Also this solution would require a more complicated communication infrastructure on the FPGA, but since this will be required anyway research is needed to determine just how much more complicated it would be.

A simpler solution to the data path issue is to create a data path for each state machine active at a particular time instead of creating the maximum at design time. This method ensures that the physical logic isn't occupied before it's needed. The communication between each state machine and its data path is determined during design time which makes it easier to design.

The system demands for the data path are:

- Must be directly connected to a NFSM clone.
- Needs to be able to interpret the state signal from the FSM made up of singular states.
- The calculations needs to be performed without being able to affect the FSM. The state transitions can't be adapted directly from the data path based on the intermediate results, it needs to go through a control system if that is needed.
- Needs to be compatible with one-hot coding.
- Needs to have adequate space to save the data during the calculation
- Must send out status signals flagging the state of the operation.

4.2.3 The Control Unit

The NFSM needs a control system to keep track of which state machine clones should be active at any time. The control system should monitor the transitions

in the NFSM and based on that make a decision on whether to activate a FSM or not.

The system demands for the control unit are:

- Has to receive the state transition signal along with the FSM clones.
- Based on the transition signal it should activate or deactivate FSMs.
- Needs to be aware of the maximum number of clones needed/available.

4.2.4 Managing Output Data

Managing the output data of the NFSM model requires some sort of mechanism for determining when the result is ready, and then receive the data. The demands of the result management system are:

- Detecting when an output is ready from a clone.
- Organizing the outputs so that they can be sent to the outside world in an orderly fashion.

Detecting when the output is ready can be solved by receiving a status signal from the data path of the FSM that is finished with the calculation. Since the FSM in this system will be detecting substrings of ones, the results of the data path will be sent to the result manager every single clock cycle.

4.3 Deterministic model

The point of this system is to create the same kind of operation as in the NFSM but without using several state machines. The DFSM implementation is quite straightforward. Simply shifting in data values continuously on an input port and using a control signal to let the state machine know when to transition could be a reasonable solution. An advantage of creating the circuit in this manner is the simplicity. Low amount of logic can also be assumed to be an advantage. The main system level challenge is coordinating the input of the control signal with the data input so that the state machine transitions at the correct time.

Figure 4.5 shows the input shift register of the data path for the DFSM version and how it is used. When the system is initialized and ready the shift register is completely empty (filled with zeros). As the input data is shifted in the values will be used in the calculations. When the ”a” value is in the first stage of the register, the result will simply be ”a” since the system then will try

to compute $result = a + 0 \cdot 0$. In the next cycle the result will be "b" since $result = b + a \cdot 0$. The first full calculation will happen when the "c" is shifted in and the result will be $result = c + b \cdot a$. This will continue as long as the system is running. Different from the NFSM since this will produce a result every clock cycle instead of waiting until all of the variables are in before outputting the first result.

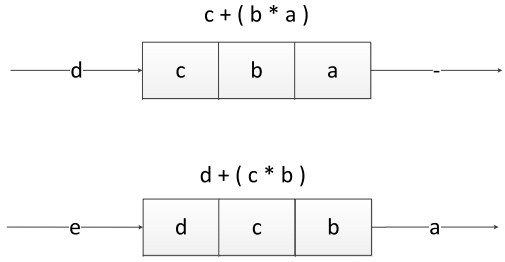

FIGURE 4.5: Input value shift register, two clock cycles

Chapter 5

Implementation examples

To get some insight into possible system-wise design issues some practical examples can be useful. Comparing the two types of implementations will also give information about space usage and timings which in turn can be interesting to compare the differences between the Deterministic Finite-State Machine version and the the Non-deterministic Finite-State Machine version. The following examples are based on the data paths defined in the previous chapter and they are written in VHDL. First a DFSM version of the modified MAC circuit can be made as an example of a traditional, simple way of implementing this kind of system. The next step is to create a model of a possible NFSM implementation. This model will not actually have the cloning capabilities that a true self-cloning state machine would have, but it will be modeled as a set of state machines that are implemented on the FPGA during design time. During run-time only the state machine "clones" that are needed will be active and the rest of the state machines will be sleeping.

5.1 Equipment and software

The software used for this project is Xilinx Integrated Software Environment (ISE) Webpack version 14.4. ISE features various tools used for designing and analyzing designs made in hardware description languages, like HDL synthesis tools, place and route tools and other FPGA configuration tools. Among these tools are a project navigator, a simulator called ISim and an FPGA Editor usually used for manually changing designs after place and route has been run. The project navigator will be used for coding both modules and test benches in VHDL, and for synthesis and place and route. ISim will be used for functional simulation and verification by observing waveforms. FPGA Editor will be used

for observing the circuits after place and route and for moving the logic around with the purpose of creating Xilinx Hard Macros.

This system will be designed for the Xilinx Virtex-4 xc4vfx12-12sf363 [Xilinx, 2008]. The FX version of the Virtex-4 has a PowerPC 405 RISC core. According to Xilinx the FX series is a "High-performance, full-featured solution for embedded platform applications." In the project the various components were designed for a Virtex-II Pro, but here it will be changed since the Virtex-4 is larger but still not too complex. Also there is no support for the Virtex-II in the newest versions of the Xilinx software. The Virtex-4 FPGA building blocks are improvements of those found in Virtex, Virtex-E, Virtex-II, Virtex-II Pro and Virtex-II Pro X product families, so the work done in the project is still relevant for this thesis. Like in the Virtex-II the Virtex-4 has four slices in a Configurable Logic Block (CLB) with fast interconnect, which again is connected to the global routing network through a switch matrix. There are 64 rows and 24 columns of CLBs in the Virtex-4, but the PowerPC takes up a significant amount of area so the number of CLBs is not as high as $64 \cdot 24$. Some specifications are given in table 5.1.

Table 5.1: Xilinx Virtex-4 Pro XC4VFX12 specifications [Xilinx, n.d.]

Resources	Number
Logic cells	12312
Slices	5,472
PowerPC Processor Blocks	1
Maximum user I/O pads	320

5.2 Deterministic model

Figure 5.1 shows the top level of the MMAC circuit when designing it in a straight-forward DFSM implementation. Table 5.2 shows an overview of the signal interface to the module. *Input_data* is the main computation data input which is a 8 bit integer that can hold values from 0 to 255. The transition signal b tells the module that the next part of the computation is to be performed. The two last input signals are a reset and a clock signal.

Output signals are *acc*, *cnt*, *done* and *valid*. *acc* is an integer used for representing the result of the computation inside the module. When the system is in the accept state a new result will be sent out via this signal each clock cycle. The *cnt* signals counts the number of times a full computation has been made. The reason for adding this signal is to make it easy to keep track of how many results we need to take care of. The *done* signal tells the outside world that the system has reach a final state, which will happen when b is set to zero, and the

system will start from the initial state in the next clock cycle. Whether or not a valid result is ready on the output is flagged by the *valid* signal. *valid* is high from the first time a result has been produced until the system is restarted. It is of interest to know that the system has started over from the beginning, and that is the reason why there is a separation between *valid* and *done*.

TABLE 5.2: MMAC DFSM signals

Name	**Direction**	**Type**	**bits**
Input_data	In	Integer	8
b	In	std_logic	1
rst	In	std_logic	1
clk	In	std_logic	1
cnt	Out	Integer	256
acc	Out	Integer	65280
done	Out	std_logic	1
valid	Out	std_logic	1

FIGURE 5.1: Top level block diagram of the MMAC circuit DFSM implementation

As can be seen in figure 5.2 the *rst* signal sets the state machine to the initial state at startup. As long as the transition signal *b* is low the FSM will remains in *S0*. However, if the transition signals becomes high the computation will start and will continue as long as the transition signal is high. From the computation state *S1* there are three possible transitions. The first choice is to go back to itself which like mentioned means that the transition signal is high and the computations should continue. If the transition signal becomes low there are two choices, ether go to the accept state *S2* or go to the fail state *S3*. The accept state sets the done signal high and then transitions to the initial state to prepare to receive another set of input data. The fail state sets all output signals low before preparing for more input. Table 5.3 shows how the output signals are set when in different states.

Table 5.3: MMAC DFSM states

State	Purpose	Acc	Cnt	Done	Valid
S0	Initial	0	0	0	0
S1	Run	Result	No. of results	0	1
S2	Final	Result	No. of results	1	1
S3	Final	0	0	1	0

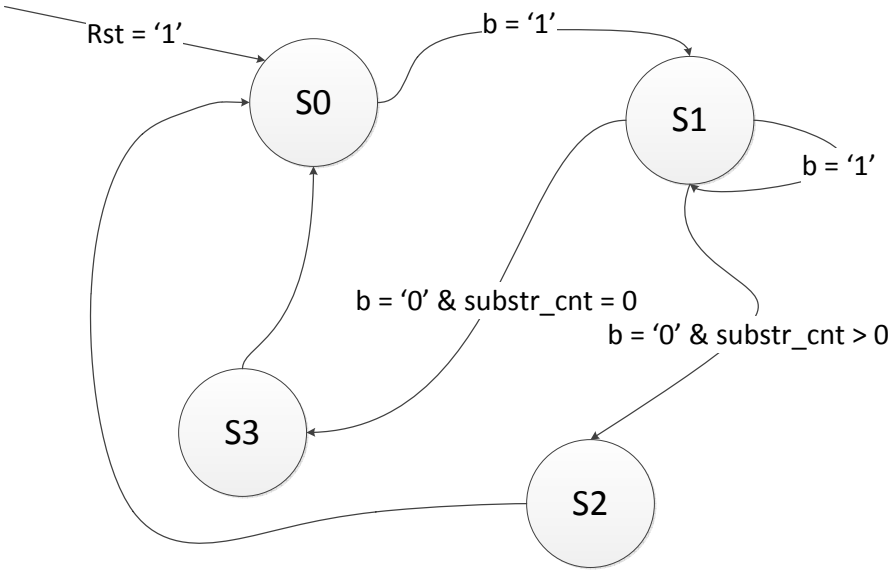

Figure 5.2: State transition diagram for the MMAC circuit DFSM implementation

5.3 Non-deterministic model

5.3.1 The singular state

The design of the NFSM model system starts by re-using the singular state defined in [Blomkvist, 2012]. Previously the singular state was defined and written in VHDL and then a Xilinx Hard Macro was created from the placed and routed singular state for it for it to be reused. When designing this system, however, implementing a previously designed Hard Macro could be very difficult and inflexible if the Hard Macro has not been designed with this particular system in mind. When the singular state was defined and designed previously it was designed for the same NFSM as it is designed for here, but it was designed to be the initial state of this NFSM exclusively and so it was very non-generic. Therefore, using the VHDL source code of the macro is a better starting point

when designing a whole system. This allows easy redesign of the state when an issue occurs, which is a great part of making the singular state as generic as possible. One could argue that if a redesign of the singular state is needed every time a new system is to be made could get tedious, but the point is to take small incremental steps in designing the singular state so that it eventually will be generic and no more redesign will be needed.

Originally the singular state was thought to need the following internal states:

- *init* - Determine if this is an initial or accept state
- *active* - The NFSM is now in this state. Should perform some action.
- *not_active* - The NFSM is not in this state. Should be idle.
- *copy_activate* - Prepare for copy, notify control that this singular state should be active in the copy.
- *copy_deactivate* - Prepare for copy, this singular state should not be active in the copy.

This previous implementation was made such that it needed to be changed for every state in every configuration since it did not have any way of being configured in run-time. This is highly impractical since each singular state then needs to be redesigned every single time in VHDL to change the enable signal logic, and then made into a new Xilinx Hard Macro. The advantage of doing it in this manner is that the singular state representation will be smaller than a more generic one. The reason the generic one will use more space is that the configuration logic will take up more space than the predetermined relations between enable signals and output status signal.

The original had states used for signaling to a control system that the singular state wanted the FSM it belonged to to be replicated. These states also had output signals associated with them which complicated the singular state interface. Simplifying the singular state makes it easier to fit it into a larger system, that is why the states *copy_activate* and *copy_deactivate* with accompanying signals is removed in this newer version. After having designed a functioning model adding some more signals to the interface can be considered.

The redesigned singular state has the following states.

- *init*
- *read_config*
- *active* - The FSM is in this state. Should be active.
- *not_active* - The FSM is in some other state. This state should not be active.

An overview of the states and transitions of the singular state can be found in figure 5.3. The *init* state waits for a signal called *conf_en* to become high for one clock cycle. When *conf_en* is detected high the internal FSM of the singular state will transition to the *read_config* state. When in the *read_config* state the singular state will receive a series of configuration vectors on the following clock cycles. The exact number of configuration vectors that will be received can be set in a constant as will be shown later. The configuration vectors contain the information required for the singular state to decide if it should be active or not at any time. What that means is that the configuration vector tells the singular state which combination of signals from other singular states connected to it and the external input/transition signal that will be required for it to transition to the active mode. So the configuration vectors contain data for each of the singular states in a FSM about which transitions are directed at the states. Each clock cycle a new configuration vector is shifted in, and a counter keeps track of how many cycles have passed to be able to know when the configuration is done. The alternative method of solving the configuration is to keep a *conf_en* signal high as long as the configuration is to happen, but that will cause problems with managing even more signals and timing the configuration signal correctly several times. If the configuration shift register is not entirely filled with valid configuration data more than one of the singular states will assume that it is an initial state and the FSM will oscillate between enabling all states at once, and disabling all states in the following clock cycle. Not all singular states in a FSM will need the same amount of configuration data since they perhaps won't have an equal amount of transitions to it. If the method of holding the *conf_en* signal high until configuration is done is used then extreme caution must be used during configuration to ensure that the configuration signal is held high just long enough, or else the previously mentioned oscillation problem will occur since more than one singular state perhaps will contain a configuration vector of only zeros in its configuration register. Using the method of counting clock cycles makes design easier since the outside world only needs to know how many spaces there are in the singular state's configuration register and don't need to worry about the timing of the *conf_en* signal except knowing to start sending in the data in the following clock cycle. If a singular states need less configuration data than there are spaces in the register then simply adding the same vector multiple times is a good way to avoid potential problems with oscillation.

Table 5.4 shows the signal interface of the singular state. The *Input* signal is an external signal which is used together with the enable signals from other singular states to determine whether or not to enable this state, which means it's the transitions signal for the FSM. The *fsm_en* signal is included to be able to suspend the operation of a FSM using singular states. Simply using the *reset* signal would require that the singular state is configured for operation again before it would be of any use, so instead the *fsm_en* is used so that the FSM can be reset while not requiring configuration. The consequence of using *fsm_en* to reset by setting it low is that all states except the initial state are set to be disabled, which in practice means that the FSM will return to the initial state

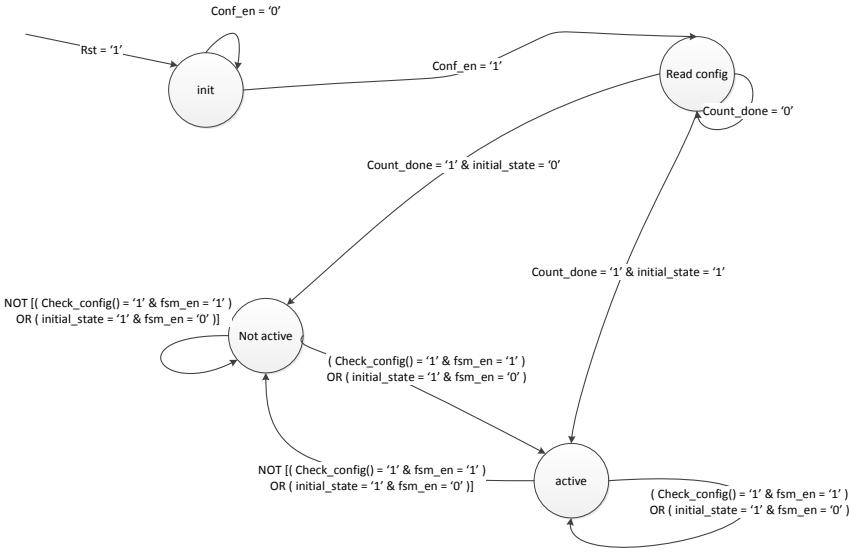

FIGURE 5.3: State transition diagram for the singular state used

and stay there ready resume operation the moment *fsm_en* is set high. The signals *en0 - en3* are the enable signals connected to the *ext_enable* signal of other singular states. *conf_en* is used for triggering the configuration.

TABLE 5.4: Singular state signals

Name	**Direction**	**Type**	**bits**
Input	In	std_logic	1
fsm_en	In	std_logic	1
reset	In	std_logic	1
clk	In	std_logic	1
en0	In	std_logic	1
en1	In	std_logic	1
en2	In	std_logic	1
en3	In	std_logic	1
conf_en	In	std_logic	1
conf_bits	In	std_logic_vector	FSM size + ext. inputs
ext_enable	Out	std_logic	1

The *active* and *not_active* states are quite similar to each other as can be seen in table 5.5. The only difference is the output value of the *ext_enable* signal.

The decision of whether to be active or not is made by using a function called *check_config* combined with detecting if the singular state is supposed to be an initial state. If a configuration vector of only zeros is detected in the *read_config* state the singular state will be enabled as soon as the configuration is done since it will either have the *fsm_en* input signal high and then use the *check_config* function to detect that no other states are active which will cause it to enable itself, or it will have the *fsm_en* input signal low but the *initial_state* internal signal high since the configuration vector was detected during configuration. Either way it will enable itself and start the running of the FSM if the *fsm_en* signal is high.

TABLE 5.5: Singular state internal states

State	**Purpose**	**ext_enable**
Init	Wait for config	0
Read_config	Config	0
Active	Enable	1
Not_active	Disable	0

The *check_config* function is part of a VHDL package called *sing_stat_package* which has been defined specifically for this purpose (Appendix A). Other than the function it has constants defining the size of the FSM the singular state is to be a part of, the names of the internal states of the singular state, the number of external input signals needed and the size of the configuration data register. These constants makes it easier to design and test the singular state since the values only need to be changed in the package file. The *check_config* function itself is supposed to match the values of the input enable signals with the vectors saved in the configuration register. A for loop iterates through the configuration register and compares the vectors with the enable signals at that time. This is a very simple albeit time consuming way of comparing the signals. An alternative could be checking the register in parallel to get a faster result, but that would require more area.

5.3.2 Interconnecting the states

The process of interconnecting the states consists of connecting the appropriate *ext_enable* output signals to the correct enable signals in the other singular states in the FSM. Since there are only four enable signals there is a limit of four states that can transition to any given state. This should be sufficient for many applications. If it isn't sufficient it's a small task to increase the number by edition the VHDL file of the singular state. After connecting the correct *ext_enable* signals to the correct enable inputs of the other singular states the configuration vectors need to be designed accordingly.

In this system it is also possible to only connect the neighboring states' *ext_enable* signals to the next state in the chain and leave the rest of the enable signals connected to either VCC or ground. The configuration vectors then need to reflect this by either using ones or zeros in the positions of the correct enable signals.

Table 5.6 shows the signal interface of a final state machine consisting of four singular states. All input signals are similar to the stand-alone singular state except for the configuration vectors which need to be unique for each state. The *state_vector* signal is a gathering of the *ext_enable* signals from the different states. If configured correctly via the configuration vectors this module will be a fully functional FSM.

TABLE 5.6: The signal interface of the connected states

Name	**Direction**	**Type**	**bits**
Input	In	std_logic	1
fsm_en	In	std_logic	1
reset	In	std_logic	1
clk	In	std_logic	1
conf_en	In	std_logic	1
conf_vec_0	In	std_logic_vector	5
conf_vec_1	In	std_logic_vector	5
conf_vec_2	In	std_logic_vector	5
conf_vec_3	In	std_logic_vector	5
state_vector	Out	std_logic_vector	4

5.3.3 Creating and connecting the data path

The data path must be designed to perform the operation of the Modified Multiply-Accumulate concept correctly using state machine input from an external state machine. Depending on a state vector input it should progress through the MMAC operation with no control over the state machine itself.

Table 5.7 shows the signal interface of the standalone data path module. *state_vector* is the signal taking in the current state from the FSM. i is the data input integer used in the operations which is continuously sent into the data path regardless of what value the state vector has. This is because of the definition of the system where a transition signal always accompanies the input data so that the operation can be done in steps, combined with the ability to always be ready for the next step in the calculation. *acc* is the output result integer which receives its value from an output register that is updated in every step in the calculation. Since two 8-bit integers should be multiplied and a 8-bit integer should be added afterwards, the maximum value should be $255 \cdot 255 + 255 = 65280$ which

means it needs to be a 16-bit integer. There is no *reset* signal since going back to the initial state is the way to set the *done* and *valid* signals low, and the FSM controls that.

Table 5.7: Data path signals

Name	**Direction**	**Type**	**bits**
state_vector	In	std_logic_vector	4
i	In	integer(0 to 255)	8
clk	In	std_logic	1
done	Out	std_logic	1
valid	Out	std_logic	1
acc	Out	integer(0 to 65280)	16

Depending on the state vector different operations are performed and different values are set on the output signals. An overview can been seen in table 5.8. When a value of *"0001"* is detected on the state vector it means that the FSM is in its initial state. In the initial state the input integer will simply be shifted into the input shift register, transferred to the output register, and then output as the result. Since the calculation has not started yet the *valid* signal is held low in this state. The *done* signal is also low since the operation is not done. When the FSM transitions to the next state and outputs the value *"0010"* the calculations begin. The output register is set to the product of the previous value of the output register and the new value of the input register. The *valid* signal is now set high to indicate that the calculation is active. The *done* signal is still low. In the next state, *"0100"*, the new value in the input register is added to the value of the output register. The rest of the signals are the same as in the addition state. Finally in the *"1000"* state the value of the output register is just set to be what it was previously. Now both *done* and *valid* are set high so that the outside world can know that a valid result is ready. When the state vector has any other value than these all of the output signals will always be zero.

Table 5.8: Data path states

State	**Purpose**	**done**	**valid**	**acc**
0001 (S1)	Idle	0	0	input
0010 (S2)	Multiply	0	1	$input \cdot acc$
0100 (S3)	Add	0	1	$input + acc$
1000 (S4)	Result	1	1	acc

Table 5.9 shows the signal interface for the whole FSM connected with the data path. This is the interface for the module that the control logic will connect to. The signal interface is a combination of the interfaces of the interconnected

states and the data path. The *state_vector* signal is not interfaced with the outside world, it is only needed for internal control. The *result* signal is connected to the *acc* signal of the data path, so it's the exact same signal with a changed name for clarification.

TABLE 5.9: Signal interface for FSM with data path connected

Name	Direction	Type	bits
transition_sig	In	std_logic	1
input_data	In	integer(0 to 255)	8
fsm_en	In	std_logic	1
reset	In	std_logic	1
clk	In	std_logic	1
conf_en	In	std_logic	1
conf_vec_0	In	std_logic_vector	5
conf_vec_1	In	std_logic_vector	5
conf_vec_2	In	std_logic_vector	5
conf_vec_3	In	std_logic_vector	5
result	Out	integer(0 to 65280)	16
done	In	std_logic	1
valid	In	std_logic	1

Figure 5.4 shows the four states connected with the data path. The *ext_enable* signals out of each of the states will connect between the states, but also to the data path. Since the FSM is of the one-hot type all of the state signals must be connected to the data path.

5.3.4 Making the control module and result register

Table 5.10 shows the signal interface of the control module. Since the FSMs the control module is supposed to control are quite simple, the control module in this system is very primitive. The control module functions by using the transition signal to determine which of the FSMs should be active and setting the values to the *en_vec* signal. By detecting if the transition signal is high or low it goes between an internal initial state and an active state. In the initial state it will just keep the first FSM enabled. In the active state it will keep enabling the next available FSM as long as there is a high transition signal by shifting in ones into a four-length shift register. If the transition signal goes low the register is cleared.

There are two processes reacting to both a change in the current state and a change in the transition signal. The first is the process controlling the next state logic, which sets the control module in the correct state depending on the transition signal. The second is the register load process which controls the

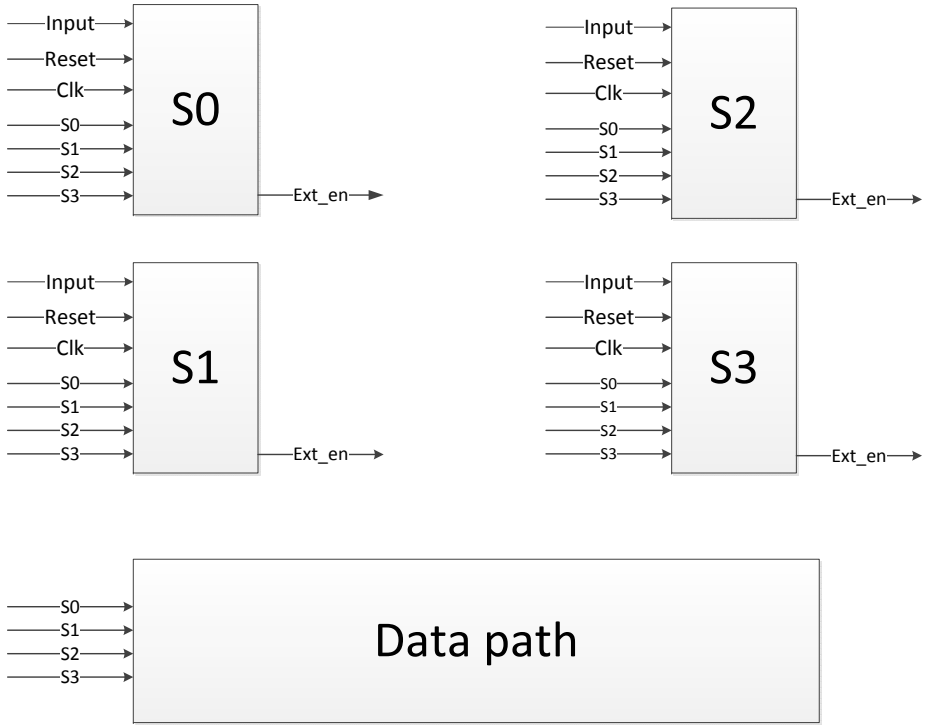

FIGURE 5.4: Top level diagram of a FSM with data path

register that enables and disables the FSMs in the system. This process needs to be able to "see the future" in the initial state in the sense that it starts shifting the register before actually being in the active state if a high transition signal is detected. That was done to make the system able to shift in values without pause and reducing timing issues with some FSMs losing an input value and several FSMs being done at the same time.

TABLE 5.10: Control module signals

Name	Direction	Type	bits
transition_sig	In	std_logic	1
done_vec	In	std_logic_vector	4
reset	In	std_logic	1
clk	In	std_logic	1
en_vec	out	std_logic_vector	4

Originally the control system was though to need the FSMs' *done* signals to determine which FSM should be active, and that is why there is a *done_vec* signal in the interface. However, for this simple design the transition signal

alone is enough information for the control system to decide when a FSM should be activated. For other (more complex) systems this might be different. Also, if the result management register is included as part of the control module the module will need to know when a result is ready to be shifted in.

Figure 5.5 shows an overview of the whole system with the four FSMs and data paths connected with the control module and the result shift register. The result register is not a part of the control module at this time, but part of the top level module connecting control with the FSMs. The register shifts in a value every clock cycle from the FSM which is in the final state at that time. The way that works is that the result from each FSM is set every clock cycle in a separate register, and when a *done* signal from one of the FSMs is set high, the value currently in the corresponding register is shifted into the result register. If none are done in a clock cycle it simply shifts in a zero. The register is filled with zeros when the system is reset.

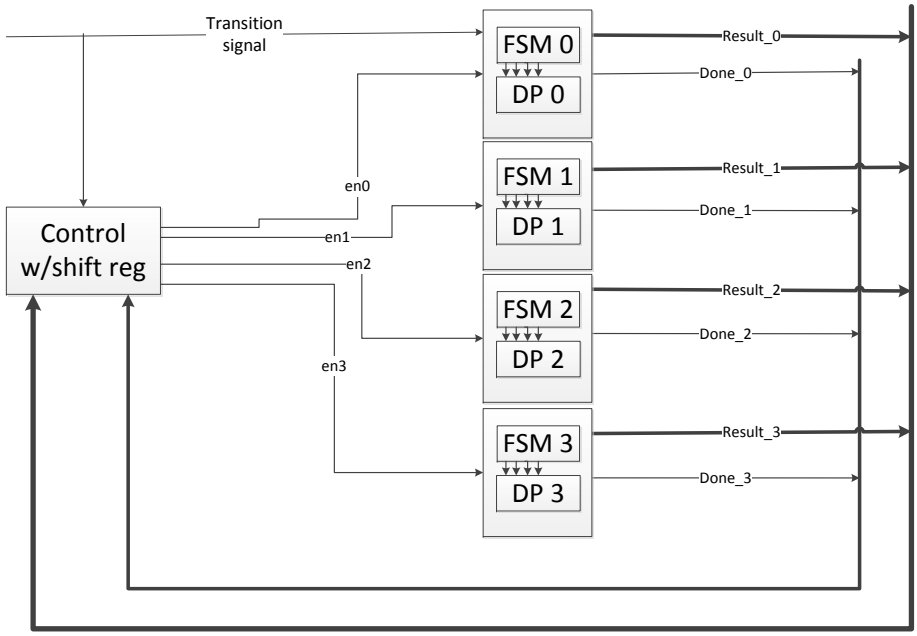

FIGURE 5.5: Top level diagram of the system

Table 5.11 shows the signal interface of the whole system. The only signal not explained before is the *conf_vecs* signal which is of type *conf_vecs*. The *conf_vecs* type is defined in the *sing_stat_package* VHDL package file and is a FSM size-length array of a *std_logic_vector*. The *std_logic_vector* size is the sum of all external inputs and the defined number of enable signals to the singular state, i.e., the size of the configuration register for the singular states. So by using the *conf_vecs* type all of the configuration vectors for the different singular states in the different FSMs can be sent into that vector.

TABLE 5.11: Signal interface for the whole system

Name	Direction	Type	bits
transition_sig	In	std_logic	1
input_data	In	integer(0 to 255)	8
fsm_en	In	std_logic	1
reset	In	std_logic	1
clk	In	std_logic	1
conf_en	In	std_logic	1
conf_vecs	In	conf_vecs	20
result_out	Out	integer(0 to 65280)	16

5.3.5 Creating Xilinx Macros

Xilinx Hard Macros are predefined circuit blocks that can be used in a design like a black box. When creating a Hard Macro one can expect that the circuit will be exactly represented in the larger system it is to be part of. The LUT configurations and other internal slice configurations, the routing, and all other aspects of the circuit will have the exact same relative position and configuration every time it's instantiated.

The reason for creating Xilinx Hard Macros is that it gives very good control over how the singular state, the whole FSM, and the FSM connected to a data path as a design unit are placed on the FPGA. Placing by hand can lead to a poorer result with regards to optimal area usage. However, making each design element as compact as possible should be a priority since they will take up a lot of space nonetheless, and having the design elements take up as little space as possible will enable more of them fit on the FPGA at the same time. Making the modules into macros will also make them easier to reuse for further development. The downsides of using Hard Macros in your design are:

- Lack of good documentation on how to design a Hard Macro both in broad terms and, more importantly, what issues one can come across and how to deal with them.
- The tools will have fewer options when trying to optimize placement and routing. What is optimal for the Hard Macro will perhaps not be optimal for the larger system it is to be a part of.
- Routing can be problematic. Clocking nets, power and ground nets can't be implemented directly in the FPGA Editor.
- The design of Hard Macros in general is very time consuming.
- The macro can be limited in how transferable it is between devices and especially between device families.

The Hard Macros are made using the Xilinx FPGA Editor and guided by a previously made tutorial [Blomkvist, 2012]. The steps taken to create a macro are:

1. Create VHDL code for the module.
2. Run synthesis and place and route.
3. Open the ncd file of the placed and routed design in the FPGA Editor.
4. Make needed changes and save as a Hard Macro nmc file.
5. Instantiate by using port mapping in the top level.

Writing the wanted behavior of the module in a HDL language is the first step. No special special steps need to be taken except perhaps making the module as small as possible to simplify later steps. Step two is running the automated synthesis and place and route tools thereby laying out the module on the FPGA. Depending on the options set in the tools this will cause the module to be spread out and use some components that are unnecessary for the Hard Macro.

Modifying the design after place and route is the most time consuming step. After performing step three and setting the design to read/write mode the modifications can begin. This task can start by removing the IOBs and replacing them with Hard Macro Ports (HMP). HMPs will be the interface to the black box, i.e., the ports the port map will connect to. Especially vectors can be tricky and it is perhaps easiest to map each individual signal in the vector to its own HMP, and then connect the individual signals to a vector outside the black box. Caution must be taken in removing the IOBs and the routing since it is easy to lose track of where the different nets were connected and what components have been removed. Clock nets cannot be manually placed and need to have a HMP at one of the clk pins of a component, and then an internal net to rest of the clk pins in the Hard Macro. In that way the external clock signal can connect to the port and clock the macro. Ground and power nets can't be placed manually either, so constant LUTs can be used to set signals constantly high or low.

5.4 Verification

Verification is, like the design itself, done step wise. The straight-forward DFSM design is verified by enabling and disabling signals on the input and sending in data. The NFSM model needs more thorough verification. Verification starts with the configuration of the singular state, then moving on to verifying that it enters the correct internal state when the correct signals are set on the inputs, connecting several of the singular states together and verifying that they

transition from one state to another correctly, connecting a data path and verifying that the registers and output data is correct, before finally connecting the control module and results register and observing if they behave correctly. Verifying the macros is done in the same manner, and the test benches of the macros use the same input signals as the test benches for the VHDL modules. All of the test benches used a clock period of 10 nanoseconds and starts off by resetting the circuit. The input data and result for the data path are written as integers in VHDL, but it is impossible to change the radix of integers in Xilinx ISim so they will always be shown as binary.

The verification of the DFSM method is done by enabling and disabling the transition signal and at the same time sending in data values. If the module sends out the correct output value it is seen to be working good enough.

The singular state is configured by setting the configuration enable signal high for one clock period. The cycle after it is set high the first configuration vector is sent in, followed by two more vectors. Filling the register with five different values is not necessary since a counter will keep the singular state in the internal state used for configuration until the configuration is done, and that means the the last value set will be taking up the last values in the configuration register and the singular state is supposed to react only to the inputs represented by those three values. When the configuration is done the singular state is to be inactive unless there is a valid input signal, or it has detected a configuration vector of only zeros in the register which means it is an initial state. Enabling different combinations of input signals lets us know if the singular state is reacting correctly. The vectors used here are "00100" (only en1 enabled), "10000" (only external input enabled) and "10010" (both external input and en2 enabled). To verify that this works correctly only 32 values need to be checked, so incrementing a vector of enable signals from 0 to 32 ensures that all possibilities are verified.

Verifying that the states transition from one another correctly is done by creating a state machine that increments from the initial state to the final state and then starts over again, like in figure 5.6. If the singular states work correctly during these simple conditions it is fine to assume that they will work for the application they are being developed for here. They are supposed to be connected as an FSM that will transition in a similar manner, but if they are to be used in a more complex FSM maybe more thorough testing should be done.

The next step is to interconnect the singular states in the same way as the NFSM model requires, and then connect a data path and see if the output values correspond to what is expected. Table 5.12 shows the configuration vectors sent into the different singular states. The vectors of State 0 makes it enable itself when there are no other states enabled, when it is enabled itself and a zero arrives on the transition input, or when the FSM has reached its final state. The last point is done for this particular application so that the FSM is "reset" and ready to yet again be enabled in the next clock cycle. The other

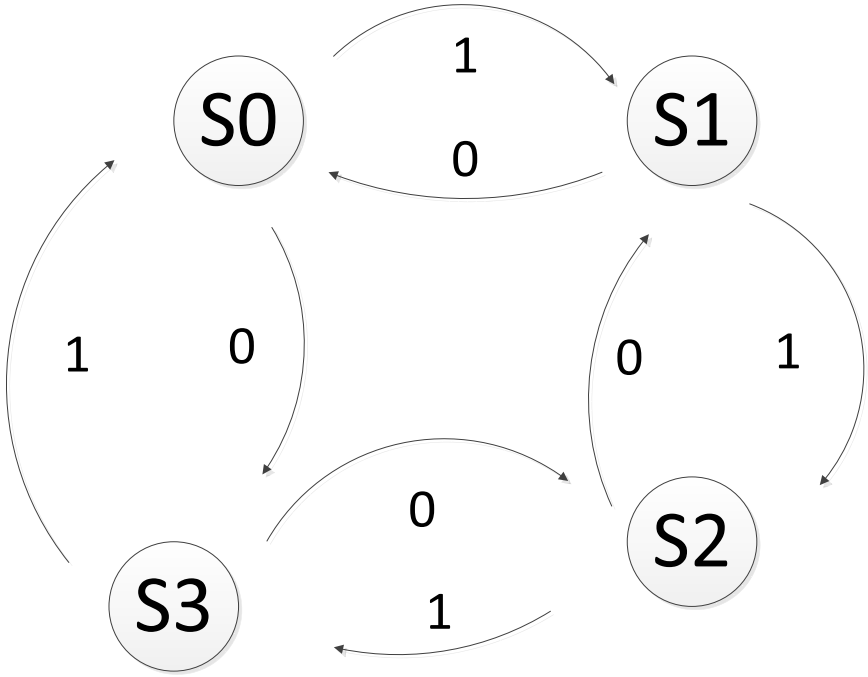

FIGURE 5.6: State transition diagram for the verification FSM for the singular states

states only need one vector, namely the vector that says it should enable itself when the previous state is enabled and a one is set on the transition signal. After the states are configured the data path is tested in the same way as the DFSM version, i.e., by toggling the transition signal and observing that the FSM goes back to the initial state and that it outputs the expected values when it reaches the final state.

TABLE 5.12: Singular state configuration vectors

	State 0	**State 1**	**State 2**	**State 3**
Vector 0	00000	11000	10100	10010
Vector 1	10000	11000	10100	10010
Vector 2	01000	11000	10100	10010
Vector 3	00001	11000	10100	10010
Vector 4	10001	11000	10100	10010

The last step is connecting the control module and results register to observe that the whole system works correctly. This is in large part similar to the previous step since the FSMs with data paths will function exactly the same.

The information gained now will be that the control module and result shift register functions as it should, which means that depending on the transition signal it will deactivate and activate the FSMs in the correct order. The result shift register should shift in the result from the FSM that sends out a done signal every clock cycle.

5.5 Implementation alternatives for a true NFSM

Reviewing a couple of different possible implementation alternatives can be useful to further assess whether or not this is a viable system. Specific details regarding the implementation will not be discussed, but determining how much area is available compared to the area usage of the circuit is especially interesting.

5.5.1 Virtual FPGA architecture

A virtual FPGA architecture was briefly presented in Chapter 3 [Hubner et al., 2011]. The architecture presented was hosted on another FPGA than the one designed for here, but a general idea of how much physical area is needed on any FPGA can be seen from the implementation results provided. For a 2-by-2 array of V-FPGA CLBs the physical requirements are over a thousand logic cells. For a 10-by-10 size V-FPGA the physical size used is about 21 thousand logic cells.

5.5.2 Framework for run-time reconfiguration

Sverre Hamre wrote a Master's Thesis in 2009 about a framework for run-time reconfiguration on Xilinx FPGAs [Hamre, 2009]. The logic needed for running the run-time reconfiguration framework takes up a very large portion of the FPGA and the reconfigurable area is only $16 \cdot 3 = 48$ CLBs large. Since every CLB in a Virtex-4 contains four slices, a maximum of 192 slices could be available if the routing allowed it.

Chapter 6

Results

6.1 Non-deterministic model

6.1.1 The singular state

Figure 6.1 shows the configuration of the singular state. The circuit is reset, then the *fsm_en* signal is held high to allow internal state transitions, then the *conf_en* signal is held high for one clock cycle. In the following clock cycles, starting at 235ns, the configuration vectors are sent in serially, with the singular state ending up with being configured for "10000", "00100" and "10010". On the fifth clock cycle after *conf_en* has been toggled the singular state starts functioning.

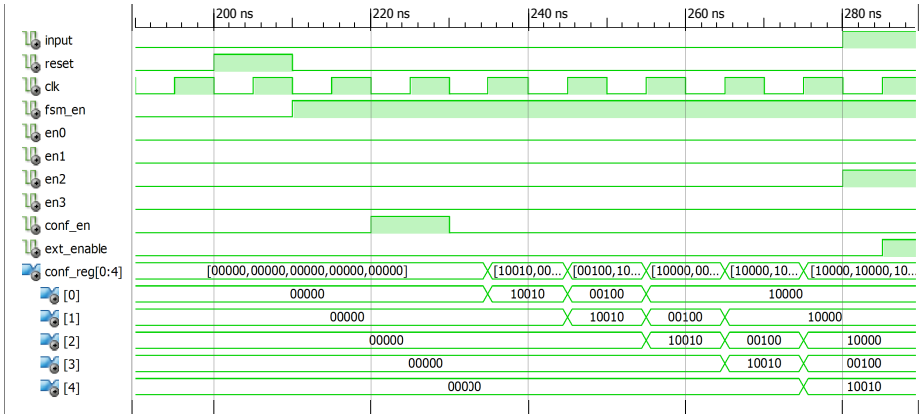

FIGURE 6.1: Singular state waveform 1

Figure 6.2 shows the singular state after configuration is done. At this point the singular state is not connected to any other states, so the enable signals are controlled by the testbench. At time 285ns the state is enabled when a "10010" is set on the inputs, at time 305ns it is enable when a "00100" is put on the inputs and finally it is enabled at time 325ns when a "10000" is set as input. In the clock cycles where the inputs have other values than the ones set in the configuration register the singular state *ext_enable* signal is low.

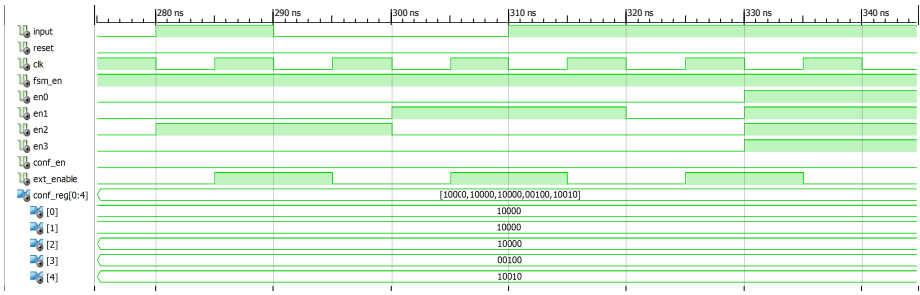

FIGURE 6.2: Singular state waveform 2

Figure 6.3 shows four singular states configured according to figure 5.6 in chapter 5. When the transition signal is low the state vector decreases and similarly when the transition signal is high the state vector increases.

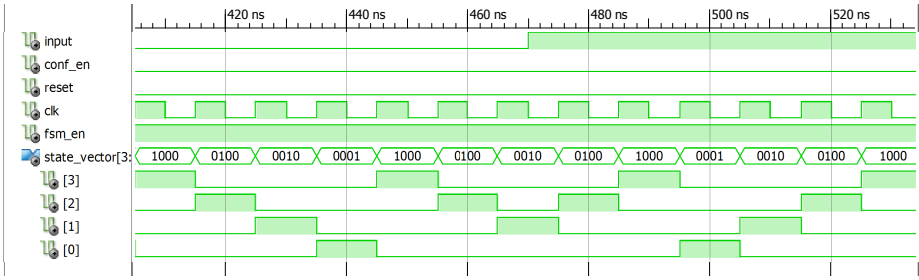

FIGURE 6.3: Singular state waveform 3

Table 6.1 shows how the physical size of the singular state increases with the an increase in the room of the configuration register. The singular state is synthesised with one-hot coding and is optimized for area.

The four states connected uses 129 flip-flops, 212 LUTs, and 108 slices.

6.1.2 Data path connected

Figure 6.4 shows the behavior of a FSM built with singular states connected as Figure 4.1 in Chapter 4 with the designed data path. The transition signal

TABLE 6.1: Logic elements compared to config reg size

Config reg size	Flip-flops	LUTs	Slices
5	33	49	28
6	38	57	32
7	43	64	36
8	48	72	41
9	54	82	46
10	59	90	50
11	64	97	54

is accompanied by input data used for the calculation. When the FSM is in the initial state the valid signal is low to indicate that the calculation is not running. When in the initial state the input register just stores new input data every clock cycle. The first result produced is a binary "10". This follows from that the input register contains a "0" the clock cycle before the transition signal goes high, so the first result is $0 \cdot 1 + 2 = 2$. The second result is $4 \cdot 5 + 6 = 26$, "11010" in binary. The result output register is not cleared when the data path goes back to the initial state which could be confusing, but it doesn't matter since both *valid* and *done* needs to be high for the result to be finished. After two calculations the transition signal goes low and the calculations stop. Figure 6.5 shows the same functionality with higher data values and continuous operation.

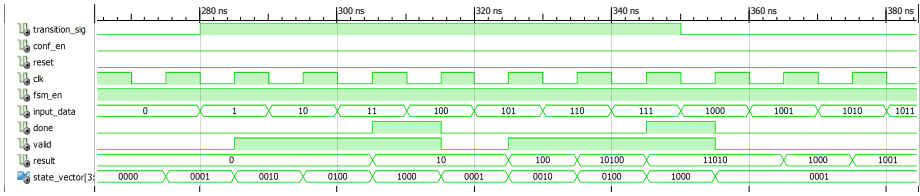

FIGURE 6.4: Data path waveform 1

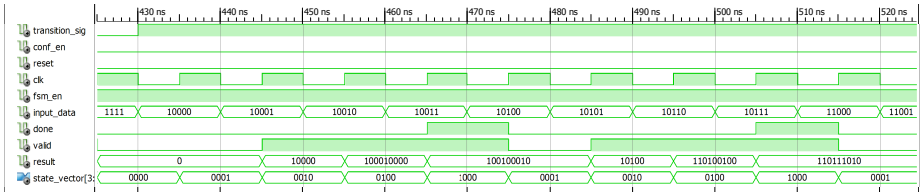

FIGURE 6.5: Data path waveform 2

Using 153 flip-flops, 298 LUTs and 155 slices, and a digital signal processing block. Removing the ability to use DSP48s causes it to use 145 flip-flops, 378 LUTs and 197 slices. Optimizing for area using one-hot.

6.1.3 Full system

Figure 6.6 shows the initialization of the whole system connected together with four FSMs including data paths for each, the control system and the result management system. The vector *en_vec* controls the enabling of the different FSMs. It starts by only having one FSM enabled and then gradually enabling one by one as long as the transition signal is high. The *done_vec* vector indicates at what time the data paths are finished processing. When the first data path is finished at time 305ns it will restart in the next clock cycle, and it has processed another result at time 345ns. The *results_out_reg* register shifts in the individual results from the different data paths and further out to the *result_out* signal. Figure 6.7 shows the *result_out* signal changing when the result register is full and starts shifting out the first result produced earlier. When the transition signal goes low all of the FSMs are disabled for one clock cycle before continuing the calculation. This is a bug related to the size of the configuration register in the singular state which will be discussed in the next chapter.

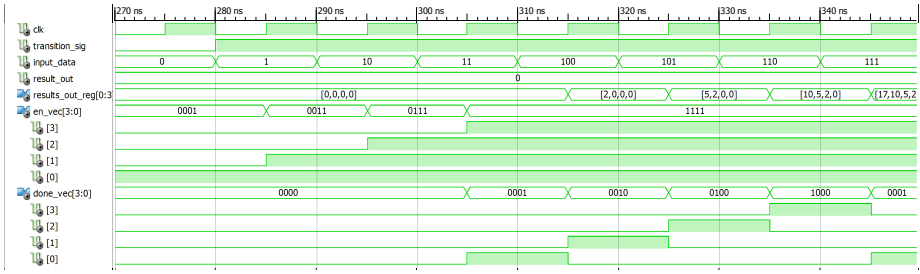

FIGURE 6.6: Full system waveform 1

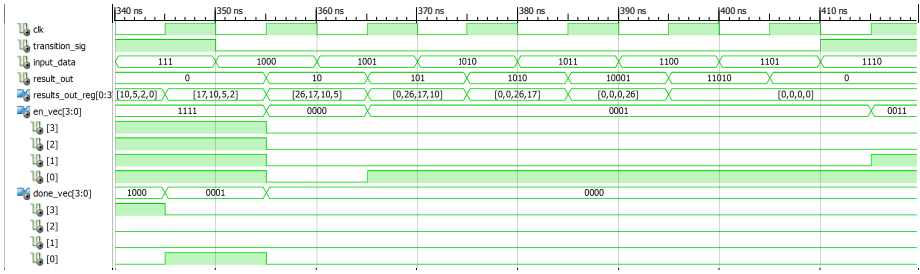

FIGURE 6.7: Full system waveform 2

Figure 6.8 shows the whole system working after having extended the configuration register of the singular state from five to seven spaces. This has eliminated the problem with having to disable all of the FSMs for one clock cycle when the transition signal goes low, and there are no timing issues.

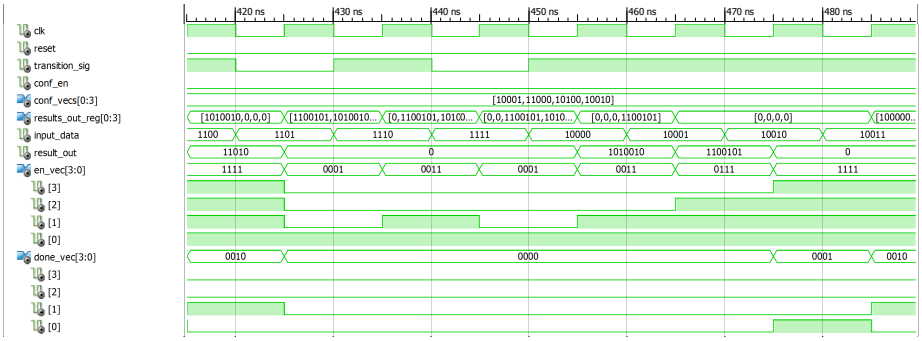

FIGURE 6.8: Full system waveform 3

6.2 Macros

The singular state macros use as many physical components as listed in Table 6.1 with perhaps very small deviations. Connecting four singular states together as a FSM will require four times as many logical components. Figure 6.9 shows a FSM with data path placed on the Virtex-4 in the FPGA Editor. This single FSM clone uses 205 slices with the data path connected.

6.3 Deterministic model

Figure 6.10 and figure 6.11 shows the waveforms of the DFSM version of the circuit. b is the transition signal and i is the input data signal. Figure 6.10 starts by showing that if the transition signal is set high for only one clock cycle, the calculation will not be completed and the system will enter the state S3. When the transition signal is high for two clock cycles the calculation will finish by entering state S2. At time 160ns the contents of the input register *reg_i* is "4,3,2". In the next clock cycle the result is $2 \cdot 3 + 4 = 10$ which is correct. The same is true for the next clock cycle, $3 \cdot 4 + 5 = 17$. So the DFSM version needs two clock cycles to start the process of enabling the results to come every clock cycle. Figure 6.11 shows the progress of the system when the transition signal is held high for a longer period.

Overview of the components for the DFSM version. Uses 34 flip-flops, 59 LUTs and 34 slices when optimized for area and using gray coding.

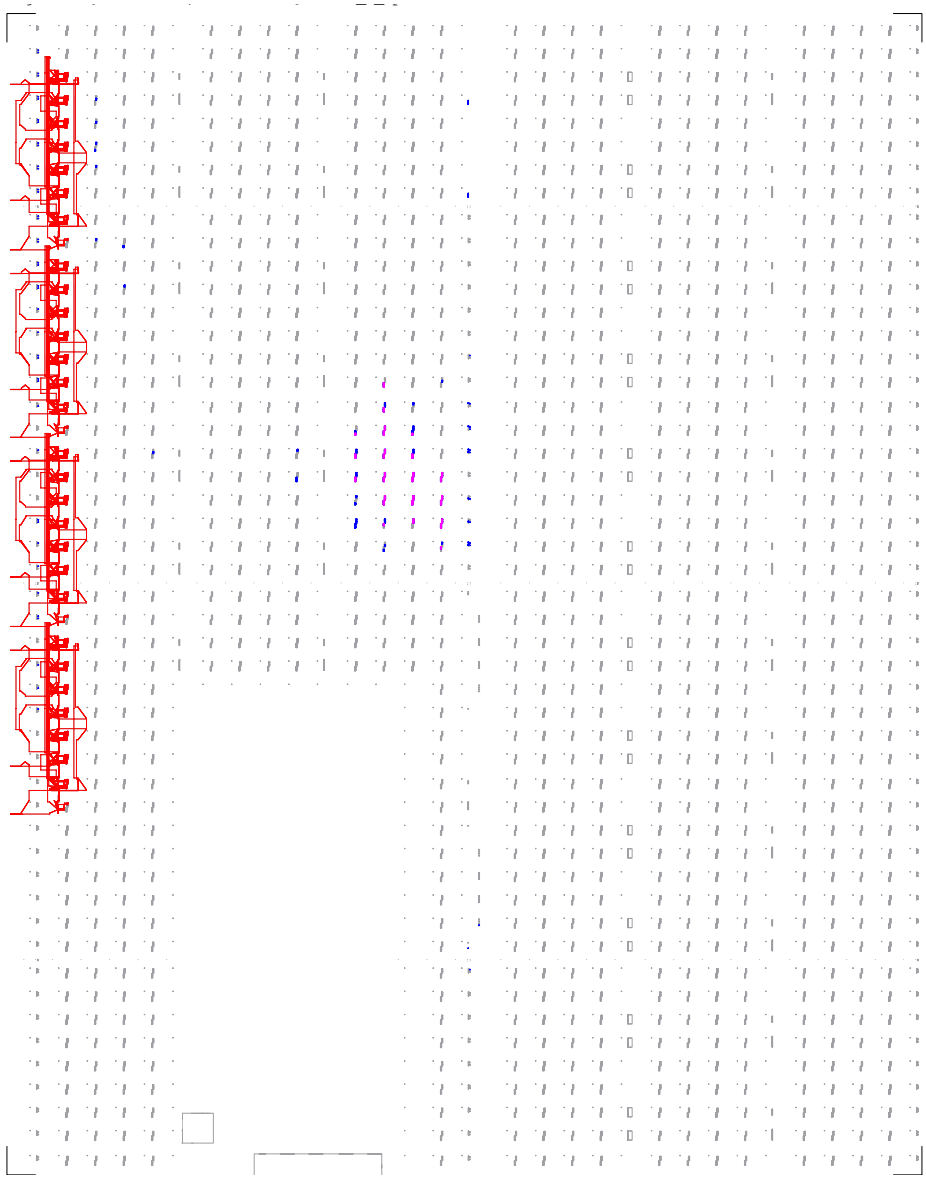

FIGURE 6.9: FPGA Editor view of a single FSM with data path

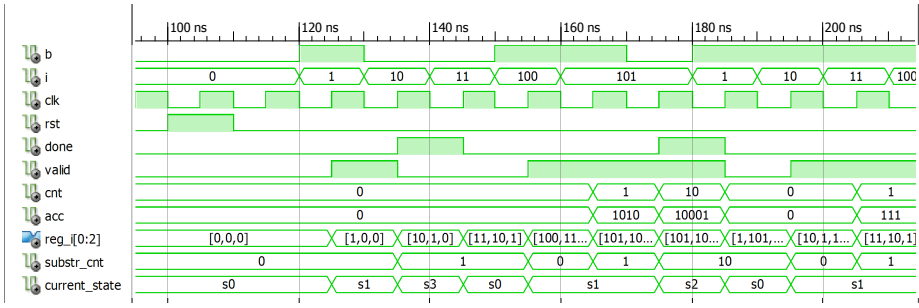

FIGURE 6.10: DFSM waveform 1

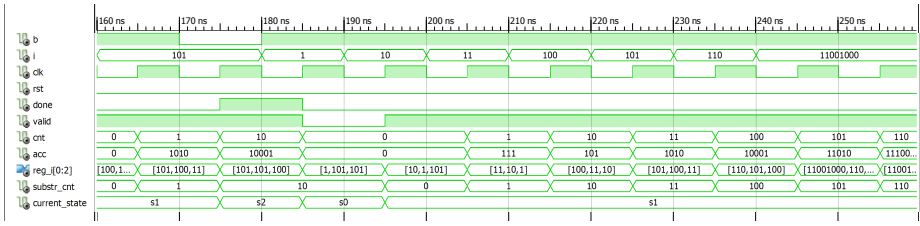

FIGURE 6.11: DFSM waveform 2

Chapter 7

Discussion

The Deterministic Model and the Non-Deterministic Model gives out the same results given the same data. The difference in functionality between the two methods is the time it takes for the data to start being output. The deterministic finite-state machine version spends two clock cycles to start outputting data while the non-deterministic version spends three clock cycles to give out the first result. After start-up they both give out one new result every clock cycle. Comparing the DFSM and the NFSM model in terms of area shows that the DFSM is clearly the better choice for this application. After creating Xilinx Hard Macros of the singular state, connecting four of those macros together as one and making a macro of the whole FSM and then connecting the proposed data path using VHDL, the NFSM model used 205 slices. That is for one single clone in a true NFSM. For the application used in this thesis four of these clones are needed to model a NFSM. That means that the circuit uses approximately four times as many components, 820 slices, not including the control system and result management system. Since the Virtex-4 designed for only has 5472 slices the NFSM model takes up a very large portion of the FPGA. The DFSM version uses 34 slices. This may not come as a surprise since letting the Xilinx tools optimize a circuit and use other types of on-chip logic like for instance digital signal processors always will be better than defining circuit elements as large blocks and simply connecting them together, but the point is to compare the scale of the difference. The MMAC is not a realistic choice of application anyway since the operation is so simple, and more complex applications like regular expression matching can give more of an advantage to the NFSM depending on the regular expression to be matched.

7.1 The singular state

The redefined singular state is a lot more generic and reusable than the one made in previous work. The flexibility comes at the price of area used for storing configuration vectors and other logic. It is now possible to use the singular state as any state in any finite-state machine with certain caveats. The first caveat is that the singular state has a limited number of enable signals, so only a limited number of other states may have transitions to the state. The second caveat is that the configuration register has a limited size. Not all possible transitions from all the connected states are possible unless the size of the configuration register is 2 to the power of the configuration vector length, and the configuration vector length is number of enables + number of external inputs. Increasing the size of the configuration register and the number of enable signals will approximately linearly increase area of the singular state, shown in Table 6.1. Given that one of the demands set in Chapter 4 was that the state should use as few logic elements as possible a careful consideration must be done during design time about how many spaces one needs in the configuration register. The configuration time increases linearly with the size of the configuration register. Loading the configuration vectors in parallel could be an option but that would increase the amount of logic further and would go against the demand that the state should be minimal. Previously the singular state needed a separate signal to determine if it was an initial state. This was changed to simplify the interface and worked well using a function checking for all zeros in the configuration vector. The simplification of the interface outweighs the area used for determining this inside the state. Determining if the FSM is in an accept state was determined to be a task better suited for the data path.

There is an error in this system related to when the transition signal goes low in the middle of a calculation. The state of the FSM will then go to "0000" since the configuration vectors for the initial states haven't accounted for a zero on the inputs when the FSM is in the second or third state (Table 5.12) , which leads to none of the states activating themselves. This is always a risk when the configuration register is not large enough to account for all possibilities. In this design the configuration register was thought to only need five spaces in the register, but it turns out it needed two more values to account for this. In a true NFSM where the clones are pruned if a zero is received this problem can possibly be avoided depending on how the pruning and cloning works, but in this model it was a problem since the intention was that one FSM should simply transition to the initial state if it detected a zero on the inputs. A method of fixing this is to design a system for flagging one or more of the enable and input signals as "don't care". In the case of this NFSM model a practical method of setting a FSM to the initial state would be to flag all of the enable signals as "don't care" when a zero arrives on the external input. What would require more area, expanding the configuration register to support all combinations of inputs or creating the "don't care" logic needs experimentation. For this model

a fix could have been making the control module more complex and adding a fifth FSM to instantly be activated when a calculation is finished in another FSM, allowing the finished FSM to prepare for one clock cycle and then putting that FSM on stand-by and so forth.

Macros were created both for a singular state and four singular states connected as FSMs to show more clearly the area usage. The singular state macro was created so that it takes up only one row of slices. Likewise when connecting the four singular states together they were placed in the same row. The reason is that one configuration frame in the Xilinx Virtex-4 is one row of 16 CLBs [Hamre, 2009], and if it is an alternative to use the on-chip reconfiguration logic fitting the macro for one clone into a single reconfiguration frame could be an advantage.

7.2 The data path

The data path performed as it should and the demands set initially was appropriate. Since the operations are so simple very few optimizations can be made. Depending on the data requirements the data register sizes can be changed. The data path has the responsibility of letting the control and result management system know that the calculation is finished. The reason is again the demand that the singular state be kept as simple as possible in addition to giving flexibility in designing different data paths for different applications.

For some reason synthesising the non-macro based singular states with the data path made it use a very large number of components compared to synthesising the whole system consisting of four times as many components. The reason was never discovered.

7.3 Control module and result register

The control module was designed according to the demands set. The only information the control module needs is the transition signal. Based on that it can enable and disable the FSMs without tracking the internal states of the FSMs. The only module that needs to track the internal states of the FSMs in this implementation is the result management shift register, which needs to know when a FSM is in the final state and therefore is ready to output a result. Actually the FSMs internal state is not directly tracked, but a signal from the data path is used to indicate that a result is ready and valid. Detecting an accept state in this manner seems like the simplest choice, letting the data path take care of the more advanced status reporting. Whether or not to integrate result management into the control module depends on the overall system structure.

In this system they were made separately with the result being in a higher level module than the control, but it is fine to do it in any other manner.

The system does not actually disable the FSMs that are finished, it simply restarts them. That would not happen in a true NFSM but it is useful here because of the way the system is designed by using a shift register to control the FSMs. If the system had used five FSMs instead of four to remedy the problem with a FSM being deactivated for one clock cycle the control system would need to handle that. In this system it could be handled by extending the control shift register in the control module and shifting in a zero every five clock cycles which would let the fifth FSM be activated at the same time as the first FSM was finished. If the system had been made in that manner it would probably represent the true NFSM better.

How to perform ϵ-moves has not been examined in this thesis. This would require a much more complicated control module which would need to be able to detect when one of the FSMs were in a state that has an ϵ-move and then clone another machine starting in the state the ϵ-move goes to. Another scenario which has not been examined is what would happen if the system is to produce additional clones going in two different directions.

7.4 Comparison with previous methods

Comparing the Self-Cloning State machine to the regular-expression matching system of Sidhu and Prasanna reviewed in Chapter 2 one can see that their method uses much less area than the Self-Cloning State Machine. Both methods use one-hot encoding to create the NFSM, but the area usage difference between the two methods is very large. They use only a 1-bit register to represent a state in the state machine, while the self-cloning state machine singular state uses at least 33 flip-flops and 49 LUTs. Clearly Sidhu and Prasannas method is better in terms of area usage.

The Sidhu and Prasanna version needs to be constructed in run-time for use with a particular regular expression. If the Self-Cloning State Machine was to be used for regular expression matching it would also need to be constructed in run-time for the same application. Configuring the states in run-time can be done in a linear amount of time depending on the configuration register size and should not have a large impact on the total construction time. Once the system is up and running they will solve the regular expression in the same amount of time given that one symbol is received every clock cycle.

Sidhu and Prasannas version is made specifically for regular expression matching and it it unclear how it would be adapted to other applications. In this thesis it has been shown that the Self-Cloning State Machine can be used for other applications, and previous work has shown that it can be used for regular

expression matching in SystemC. Nonetheless, it is difficult to say if the Sidhu and Prasanna version is any less adaptable than the Self-Cloning State Machine.

7.5 Choosing the run-time reconfiguration solution

The two run-time reconfiguration options, the V-FPGA and the framework made by Sverre Hamre, both have very little area available. Assuming the area is plentiful on a FPGA the methods can be compared in other ways. Both methods lack tool support so it is difficult to implement the system on both of them. Because the V-FPGA will be much slower in execution than the FPGA it is implemented on the method of Hamre may be the fastest in the execution itself. However, reconfiguring the Xilinx FPGA in run-time may not be much faster than doing it on the V-FPGA. More research is needed to compare the reconfiguration times of the two methods.

The main advantage of using the V-FPGA architecture is that it can be device independent. The advantage from device independence is not only from the time consumed for developing the Self-Cloning State Machine for each individual FPGA, but also that it can bring run-time reconfiguration to devices that do not support it and from avoiding device-specific primitives which otherwise must be avoided. Among the problems eliminated are; interfacing between multiple clock networks, avoiding the mirroring of logic between the top and bottom halves in the Virtex-4 architecture, avoiding having to design around other types of logic like DSPs and RAMs. These are all taken care of by the new abstraction layer. An additional advantage is a smaller chance of ruining the FPGA if an error is made during reconfiguration, which can happen if the on-chip primitives are used.

Chapter 8

Conclusion

Comparing a deterministic and non-deterministic state machine implementation of the modified multiply-accumulate circuit shows that the DFSM implementation is vastly superior in terms of area, slightly quicker to give outputs and does not require any initial configuration. Comparing these two implementations is not really fair since the application developed for here was chosen because of the simplicity in detecting errors, but nonetheless it shows the scale of the difference between an ordinary DFSM implementation and a NFSM implementation.

A new definition of the singular state for use in a Self-Cloning State machine has been made. This new definition has been shown to be able to be set into any position in any one-hot coded finite-state machine. The configuration of the singular state is relatively easy to perform by using a vector representing the individual external inputs and enable signals from other singular states. This singular state takes up a large area on the FPGA comparing to a state in normal one-hot coded state machine because of the configuration logic added. The state can determine if it is an initial state in a state machine by using the configuration vectors alone. Problems were discovered related to the configuration register and two different methods of fixing those were discussed.

The data path, control module and result register of the NFSM implementation performed the task as expected. The data path was well suited to show how the NFSM data flow will be in a true implementation. The control system using a shift register was an effective and simple way of enabling and disabling the state-machines in run-time. Using a fifth state machine as a stand-by for when a result was ready could have been an alternative to expanding the configuration register when addressing the issue of the FSM not restarting immediately when finished with a result. Using status signals from the data path connected to a FSM was a good way to control the flow of results.

Comparing the Self-Cloning State Machine to the Sidhu and Prashanna implementation of a NFSM for regular expression matching shows that the Self-Cloning State Machine uses a lot more area. Given that the other implementation aspects are quite similar, i.e., having to construct the NFSM in run-time and performing the regular expression matching itself, the area is the most important factor to use to determine if this is viable. Therefore the Sidhu and Prasanna method of implementing this is determined to be better than the current implementation model of the Self-Cloning State Machine.

The V-FPGA method of implementation for a true NFSM seems to be the best solution. The reason is the low chance of destroying the FPGA during reconfiguration, the device independence that can be achieved by using V-FPGAs, and not having to worry about the low level physical architecture of the FPGA chosen. The macros created are specific for the Virtex-4 family of FPGAs so they will not work here and some other solution must be chosen to restrain the area.

8.1 Future work

- More experiments on balancing the area of the singular state with the number of needed enable signals.
- Expanding the singular state with ”don’t care” logic.
- Research into how to handle ϵ-moves.
- General research into more complex NFSMs than the one modeled here to determine more system requirements.
- Implement a virtual FPGA architecture on a very large FPGA and attempt to create a true Self-Cloning State Machine by using the singular state.

Bibliography

ARM [n.d.], 'Advanced microcontroller bus architecture', `http://www.arm.com/products/system-ip/amba/amba-open-specifications.php`.

Ashenden, P. J. [2008], *Digital Design: An Embedded Systems Approach Using VHDL*, Elsevier.

Bailey, D. G. [2011], *Design for embedded image processing on FPGAs*, first edn, John Wiley & Sons (Asia) Pte Ltd.

Blomkvist, D. H. [2012], 'Self-Cloning State Machines on FPGA'.

Compton, K. and Hauck, S. [2002], 'Reconfigurable computing: a survey of systems and software', `http://doi.acm.org/10.1145/508352.508353`.

Hamre, S. [2009], Framework for Self Reconfigurable System on a Xilinx FPGA, Master's thesis, Norwegian University of Science and Technology.

Harvey, B. [1997], *Computer Science Logo Style Volume 3*, second edn, MIT Press.

Hubner, M., Figuli, P., Girardey, R., Soudris, D., Siozios, K. and Becker, J. [2011], A heterogeneous multicore system on chip with run-time reconfigurable virtual fpga architecture, *in* 'Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on', pp. 143–149.

Koch, D., Torresen, J., Beckhoff, C., Ziener, D., Dennl, C., Breuer, V., Teich, J., Feilen, M. and Stechele, W. [2012], Partial reconfiguration on FPGAs in practice - tools and applications, *in* 'ARCS Workshops', pp. 297–319.

Lagadec, L., Lavenier, D., Fabiani, E. and Pottier, B. [2001], Placing, routing, and editing virtual fpgas, *in* 'Proceedings of the 11th International Conference on Field-Programmable Logic and Applications', FPL '01, Springer-Verlag, London, UK, UK, pp. 357–366.
URL: *http://dl.acm.org/citation.cfm?id=647928.740044*

Plessl, C. and Platzner, M. [2004], Virtualization of Hardware - Introduction and Survey, *in* 'Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'04)', Las Vegas, Nevada, USA, pp. 63–69.

Sidhu, R. and Prasanna, V. K. [2001], 'Fast regular expression matching using FPGAs', `http://dl.acm.org/citation.cfm?id=1058885`.

Svarstad, K. and Volden, K. [2011], Replicating non-deterministic finite state machines as a mechanism for run time reconfiguration on FPGAs.

Volden, K. [2011], Compiling Regular Expressions into Non-Deterministic State Machines for Simulation in SystemC, Master's thesis, Norwegian University of Science and Technology.

Xilinx [2008], 'Virtex-4 FPGA user guide ug070 (v2.6)', `http://www.xilinx.com/support/documentation/user_guides/ug070.pdf`.

Xilinx [n.d.], 'Virtex-4 family overview ds112 (v3.1) august 30, 2010', `http://www.xilinx.com/support/documentation/data_sheets/ds112.pdf`.

Appendix A

Code

A.1 Package file

```
1  library IEEE;
   use IEEE.STD_LOGIC_1164.all;
3
   package sing_stat_package is
5    constant FSM_SIZE : integer := 4; -- Size of FSM
     constant EXT_INPUT_NUM : integer := 1; -- Number of external
       inputs
7    constant EN_NUM: integer := FSM_SIZE + EXT_INPUT_NUM; -- number
       of enable signals in the singular state
     constant CONF_DATA_LEN: integer := 5; -- Size of config data
       register in each singular state.
9
      type state_type is (init, read_config, active, not_active);
11
     subtype conf_info is std_logic_vector(EN_NUM - 1 downto 0);
13   type conf_data is array(integer range 0 to CONF_DATA_LEN-1) of
       conf_info;
     type conf_vecs is array(integer range 0 to FSM_SIZE - 1) of
       conf_info;
15
   function check_config(signal valid : conf_data; signal input :
       conf_info) return std_logic;
17
   end sing_stat_package;
19
   package body sing_stat_package is
21   function check_config(signal valid : conf_data; signal input:
       conf_info) return std_logic is
       variable hit: std_logic;
23     begin
         hit := '0';
25       for i in 0 to (CONF_DATA_LEN - 1) loop
           if (valid(i) = input ) then
```

```
27            hit := '1';
          end if;
29        end loop;
          return hit;
31    end check_config;

33 end sing_stat_package;
```

./Appendices/code/sing_stat_package.vhd