# A Mix-Grained Architecture for Improving HLS-Generated Controllers on FPGAs

by

Shadi Assadikhomami

B.Sc., Sharif University of Technology, 2013

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

**Master of Applied Science**

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Electrical and Computer Engineering)

The University Of British Columbia
(Vancouver)

June 2017

# Abstract

With the recent slowdowns in traditional technology scaling, hardware accelerators, such as Field Programmable Gate Arrays (FPGAs), offer the potential for improved performance and energy efficiency compared to general purpose processing systems. While FPGAs were traditionally used for applications such as signal processing, they have recently gained popularity in new, larger scale domains, such as cloud computing. However, despite their performance and power efficiency, programming FPGAs remains a hard task due to the difficulties involved with the low-level design flow for FPGAs. High-Level Synthesis (HLS) tools aim to assist with this time-consuming task by supporting higher level programming models which significantly increases design productivity. This also makes the use of FPGAs for large scale design development for evolving applications more feasible.

In this thesis we explore the potential of modifying the current FPGA architecture to better support the designs generated by HLS tools. We propose a specialized mix-grained architecture for Finite State Machine (FSM) implementation that can be integrated into existing FPGA architectures. The proposed mix-grained architecture exploits the characteristics of the controller units generated by HLS tools to reduce the control-path area of the design. We show that our proposed architecture reduces the area of the next state calculation in FSMs by more than 3X without impacting the performance and often reducing the critical path delay of the next state calculation in FSMs.

# Lay Summary

Programming low-level, dedicated hardware computing systems, such as Field-Programmable Gate Arrays (FPGAs), is more challenging and time consuming compared to programming higher-level software for general-purpose processors. Despite the difficulties associated with programming hardware, FPGAs still remain an appealing solution over general-purpose processors for many applications due to their higher efficiency. High-Level Synthesis (HLS) aims to ease the hardware programming by enabling the use of higher-level software languages to program FPGAs. However, there is generally a trade-off between programmability and efficiency when using HLS tools, which can often result in a less efficient hardware design than programming FPGAs using low-level programming languages. In this dissertation, we aim to narrow the gap between programmability and efficiency when programming FPGAs using HLS tools. We propose a novel modification to current FPGA architectures that exploits common properties of HLS-generated designs to improve the FPGAs efficiency by reducing the total area of the hardware design.

# Preface

This dissertation is based on a research project conducted by myself under the supervision and guidance of Professor Tor M. Aamodt. I assisted with defining the problem space and was responsible for identifying challenges within this problem space, and designing and modelling the architecture to evaluate the proposed solution. I also conducted the experiments and collected all of the data represented in this dissertation, except the data shown in Section 3.1.4 which has been collected by Jennifer Angelica with my assistance under the supervision of Professor Aamodt.

# Table of Contents

# List of Tables

# List of Figures

# List of Abbreviations

**CPU** Centeral Processing Unit

**GPU** Graphic Processing Unit

**DSP** Digital Signal Processor

**ASIC** Application-Specific Integrated Circuit

**HLS** High-Level Synthesis

**FPGA** Field Programmable Gate Array

**CGRA** Coarse Grain Reconfigurable Architecture

**FSM** Finite State Machine

**CDFG** Control/Data Flow Graph

**DCG** Directed Cyclic Graph

**RAM** Random Access Memory

**LUT** Look-Up Table

**HDL** Hardware Description Language

**RTL** Register-Transfer Level

**CAD** Computer Aided Design

**SOC** System On a Chip

# Acknowledgments

I would like to thank my supervisor, Professor Tor M. Aamodt, for the valuable guidance, support, and insight he provided during these years and for giving me the opportunity to do high-quality research. This work would have not been possible without him. I also gratefully acknowledge the funding provided by the Natural Sciences and Engineering Research Council of Canada (NSERC) that made my research possible.

I would like to thank everyone in the computer architecture group at UBC, including the graduate students and undergraduate interns, who I had the opportunity to work with. I would also like to thank my former lab-mates, Tim Rogers, Ahmed ElTantawy, and Andrew Boktor, and my friends from the SOC group, Fatemeh Eslami and Hossein Omidian, for their kind help, advice, and sharing their knowledge and experience with me. I would like to specially thank my best friend and a senior PhD students in our group, Tayler Hetherington, for his help, support, feedback, and everything I have learned from him during these years.

And last but not least, I would like to thank my family who taught me to never give up and fight for my goals, and for their invaluable support throughout my life.

*To my parents*

# Chapter 1

# Introduction

Since their emergence, computing systems have undergone a series of revolutionary improvements in their performance, energy efficiency and cost-effectiveness. These improvements were achieved by architectural innovations and advancements in the semiconductor industry. Advancements in semiconductor technologies provide large improvements to computing systems by drastically increasing the amount of processing capability per unit of area and power. Historically, these advancements have followed Moore's law [22], which states that the number of transistors on a chip will double approximately every two years and Dennard Scaling [8], which states that the power density of transistors remains constant as their size scales down which enables smaller and faster transistors. However, in recent years, Moore's law and Dennard scaling have slowed, resulting in diminishing returns from semiconductor improvements.

Additionally, to broaden the scope of applications able to benefit from such computing systems, architectures were designed with generality in mind, such as the CPU. However, due to the slowdowns in the rate of improvements for computing systems, there has been a shift towards using alternative architectural designs and specialized hardware accelerators to keep up with the growing computational demands of today's applications.

Hardware accelerators are customized circuits that are designed for performing a particular set of tasks [28]. They have shown great potential to improve the performance and energy efficiency of applications by eliminating the overheads that

come with having a more general purpose architecture. Graphic Processing Units (GPUs), Application-Specific Integrated Circuits (ASICs), Digital Signal Processors (DSPs), and Field Programmable Gate Arrays (FPGAs) are examples of the most common hardware accelerators [28].

These accelerators range in their level of specialization and programmability. Similar to CPUs, GPUs offer a high degree of programmability, however, are designed to accelerate a class of applications with large amounts of data-level parallelism. In contrast, ASICs are designed to perform a specific set of tasks with dedicated hardware at the cost of little to no programmability. FPGAs bridge the gap between programmable processors and dedicated hardware accelerators by providing a reconfigurable and programmable hardware platform. FPGAs improve the flexibility over ASICs, while maintaining a portion of the improvements in performance and energy efficiency of a hardware design compared to a general purpose architecture.

More recently, FPGAs have been gaining popularity in domains they have not typically been used for, such as cloud computing [[39], [38]]. Some of of the world's biggest datacenters, such as Microsoft and Baidu, are now deploying FPGAs in their servers [[44], [23], [6]], and Amazon is now offering FPGA cloud instances in the amazon web services[3]. Additionally, with the acquisition of Altera by Intel in 2015 [13], FPGAs may become more closely tied to general purpose architectures, making them more accessible and increasing the use in new markets, such as Cloud computing.

FPGAs are traditionally programmed using hardware design languages (HDLs), such as Verilog or VHDL. Hardware design is notoriously more difficult compared to software development. This is one of the main issues with using FPGAs for accelerating large scale applications. However, recent advances in high-level synthesis (HLS) significantly increase the productivity of hardware design by enabling the designers to use higher level software programming languages, such as C/C++ and OpenCL, which makes FPGAs easier to use for accelerating larger scale applications. Therefore, HLS is now becoming a part of the main hardware design flow [[7], [42]]. This raises the question - can we modify the FPGA architecture and CAD flow such that they can be more efficiently used by HLS tools? In this dissertation we aim to answer this question by exploring the potential of improving

the architecture of FPGAs to better tune them for HLS design flow by analyzing the characteristics and requirements of designs generated by HLS tools.

## 1.1 Motivation

FPGA architecture consists of an array of generic programmable logic blocks and programmable routing switches that enables them to implement any logic function. This flexibility comes with the cost of area, performance, and power overhead that causes the FPGAs implementation of a given design to be at least an order of magnitude larger than the ASIC implementation, with a critical path delay ratio of about 3 to 4 [14]. To bridge this gap, FPGA designers have introduced hard blocks such as mutiplier/accumulator, block memories, and floating point units to modern FPGA architecture to mimic efficiency of ASICs for a common set of operations[17]. Hard blocks are ASIC-like hardware units that are less programmable, but more efficient than programmable logic blocks. Despite their efficiency improvements, the area of underutilized hard blocks is wasted, therefore, the hard block architecture must consist of function units and logic operations that are commonly used among a representative set of important FPGA applications.

The existing hard blocks on FPGA architectures have been designed to accelerate the operations that are common among the original application domains that were using FPGAs. However, the recent shift to use FPGAs in new domains with varying processing requirements raises the question - are there other common operations among these new application domains that can benefit from being mapped to hard blocks? The same question can be asked regarding the recent increased popularity in the use of HLS tools - Due to their automated nature to generate hardware designs, as opposed to a human hardware designer approach, is it possible that they generate any special structure in hardware that can be exploited by new hard blocks?

In this work, we aim to answer this question by studying the controller unit hardware generated by HLS tools. HLS tools often generate large explicit controller units that are modelled by finite state machines. These control units can have a big influence on the total area of the design in cases where the realization of the data path requires a large number of states and control signals [20].

In this dissertation we analyze the characteristics of the finite state machines that are generated by HLS tools and argue that these state machines all share common behaviours that can be exploited to design an alternative hardware implementation for such FSMs. We evaluate our proposed architecture by detecting and extracting the FSM as a standalone circuit from the application and compare it against the baseline FSM implemented purely in FPGA soft logic. We show that our proposed architecture has a great potential to reduce the area implementation of the next state generation logic in FSMs as well as reducing its critical path delay.

## 1.2 Contributions

This thesis makes the following contributions:

- Identifying common characteristics among state machines generated by HLS-tools.
- Proposing a novel architecture to improve area efficiency of next state calculation logic in FSM implementation without affecting performance.
- Proposing a novel state encoding technique which exploits certain properties of HLS-generated FSM described in Section 3.1.2.
- Evaluating the the area and delay improvement of the proposed architecture compare to a baseline FPGA architecture which shows an average area reduction of 70% as well as critical path delay reduction of 45%.

## 1.3 Organization

The rest of this dissertation is organized as follows:

- Chapter 2 first details the background FPGA architecture used in this dissertation. It then provides the necessary background on finite state machines as an important part of digital systems.

- Chapter 3 performs analysis on FSM characteristics and presents a novel configurable architecture to improve the area efficiency of FSM implementation on FPGAs.
- Chapter 4 presents the methodology used to implement and evaluate the designs introduced in Chapter 3.
- Chapter 5 evaluates the area/delay improvements of the proposed specialized FSM architecture.
- Chapter 6 discusses related work.
- Chapter 7 discusses directions for potential future work.
- Chapter 8 concludes the dissertation.

Next section describes the necessary background for this work.

# Chapter 2

# Background

This chapter presents the necessary background information to understand the contributions of this Thesis. First, this chapter describes the architecture of contemporary FPGAs. It then discusses the required background to understand the finite state machines and their traditional implementation methods on FPGAs. Finally, this chapter provides a brief summary of the standard hardware design flow and the addition of High-Level Synthesis (HLS) tools to this flow.

## 2.1 FPGA Architecture

A traditional FPGA architecture consists of an array of generic logic blocks that are connected via configurable routing channels. The main components of these logic blocks are n-input (normally 6-input) Look-Up Tables (LUTs), small one-bit hard adders, and optional flip-flops that enable registering the output of the block. A n-input LUT can be configured to implement any logic function that maps the n-bit input to a 1-bit output. Therefore, using LUTs in logic blocks turns them into generic flexible blocks that are capable of implementing any logic function [15].

As discussed in the Chapter 1, in modern FPGA architectures some of these generic blocks are replaced by hard blocks such as multiply-add, floating point operations, and memory blocks to improve the efficiency of these specific set of operations [14]. This is shown in Figure 2.1.

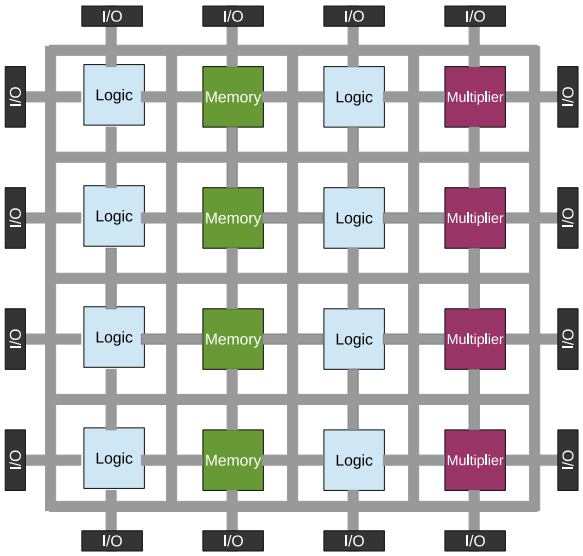



**Figure 2.1:** Basic FPGA architecture [15]

## 2.2 Finite State Machines

In this section we present background information for Finite State Machines (FSMs) as an important part of digital circuit design and review the alternative methods of implementing FSMs on FPGAs.

### 2.2.1 Finite State Machine Definition and Representation Models

Logic circuits consist of two main parts: data-path and control-path. The control-path is also known as the control unit.The general block diagram of digital circuits is shown in Figure 2.2. The data-path can be described as functional units that perform the computational tasks (data operations) in an application [21]. The control unit, on the other hand, generates the control signals required to direct the operation of data-path according to the timing constraints, data, and control dependencies in an application. Finite state machines are a common way to describe the control path in logic circuits. As the name suggests, an FSM is composed of a limited set of states and the corresponding transitions between these states. Each

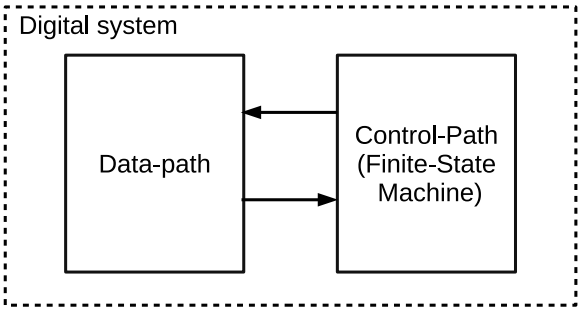



**Figure 2.2:** Digital systems structure

state corresponds to a specific state in the real design. The transition between these states happens based on the *current state* of the system and the set of inputs to the FSM. Each states has a set of associated control signals that are dependant on the current state of the system and, potentially, the input signals. In a Moore FSM the output signals are defined only based on the current state of the system, where as in a Mealy model both inputs and current state are used to determine the value of output signals.

### State Transition Table

A state transition table is one of the common ways of representing an FSM. The state transition table is a truth table, where the inputs and current state form the input column of the table, while the output column contains the next state value and outputs of the FSM. It is a simple method to define the state transitions and the values of output signals based on the current state and inputs.

### State Transition Diagram

The state transition diagram is the equivalent graph-based representation of the state transition table [21]. A state transition diagram is a directed cyclic graph (DCG) $G = (V, E)$ where each vertex $v_i \in V$ represent a unique state and each edge $e_{ij} \in E$ shows a transition from the corresponding state $v_i$ to the $v_j$. The edge labels indicates the input sequence that causes the corresponding transition. Depending on the FSM model, Mealy or Moore, the output of each states will be either part of

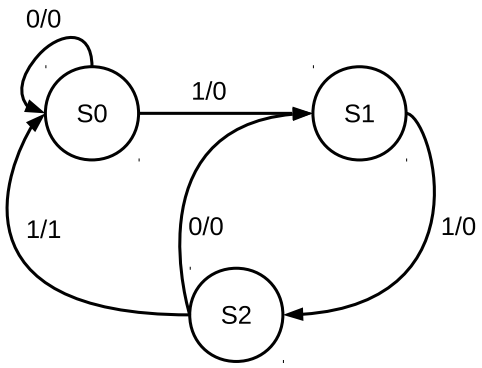



**Figure 2.3:** The state transition graph of a Mealy FSM.

the edge or vertex label respectively. This is shown in Figure 2.3

### 2.2.2 Finite State Machine Implementations

There are two main approaches to implement FSMs on FPGAs, which are discussed below.

#### LUT-Based Implementation

A LUT-based implementation is the common conventional way to implement FSMs on FPGAs. Figure 2.4 shows the block diagram of an FSM. It consists of state registers to hold the current state value, and combinational logic to calculate the next state value and output signals. The combinational logic is implemented using FPGAs' LUT-based logic blocks. However, the flexibility of LUTs to implement any logic function comes at cost of increased area, power, and performance. Logic minimization algorithms and state assignment techniques are used to find the optimal combination circuits, which realize the state transfer function and output function [21].

#### RAM-Based Implementation

After embedded block RAMs were introduced to FPGA architectures, a large body of research investigated the benefits of using block RAMs as an efficient method

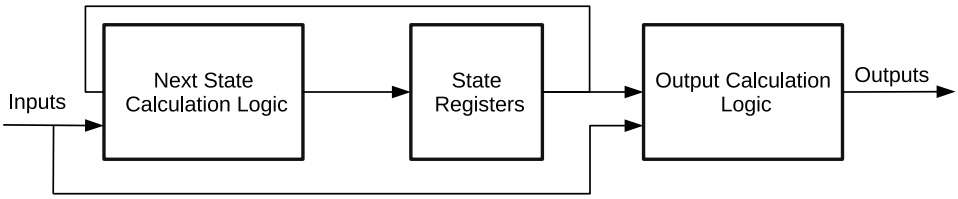



**Figure 2.4:** General structure of finite state machines

for implementing FSMs [[26], [34], [30], [31]]. RAM-based FSM implementations have a great potential to reduce the area usage by utilizing less of the FPGA's routing and logic resources, which consequently improves the area and power consumption of the design. Figure 2.5 shows an example of a RAM-Based FSM implementation. In this example, the FSM has $q$ inputs, $r$ outputs, $p$ states, which requires a n-bit encoding. The state value will be stored in a n-bit register and together with the input, form the address to the memory unit to look up the value of the next state and output signals. Such a memory unit will have $2^{(n+q)}$ entries of size $(n+r)$ to accommodate the next sate and output values for all the combinations of current state and input values. However, one potential problem with such implementation is the exponential growth in memory size with an increase in number of states and inputs. For the scenario where there are several inactive inputs at each states that do not contribute to the next state calculation, a potential solution has been proposed that utilizes a selecting mechanism to choose the active inputs at each state to address the memory locations in order to avoid the unnecessary increase in the memory size [10].

## 2.3 Hardware Design Flow

Programming hardware tend to be more difficult compared to software development. The traditional hardware design flow requires designers to use low-level hardware description languages such as Verilog and VHDL, to directly describe a given high-level algorithm. This description is typically at register transfer level (RTL) where a circuit is described by its logic operation, registers, and their corresponding data flow. The RTL design will then be mapped to an FPGA using the Electronic Design Automation (EDA) tools , which after synthesizing the RTL de-

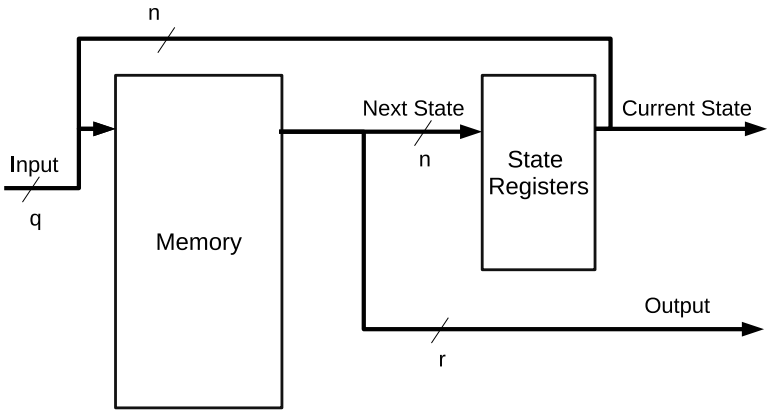



**Figure 2.5:** Memory-based implementation of FSMs

sign into a gate-level netlist and applying the logic optimization techniques, try to map the design onto an FPGA architecture in an iterative manner.

The very low-level nature of RTL design, various design constraints and requirements, and long EDA process makes the hardware design a very challenging and time-consuming task compared to the typical sequential programming in software. On the other hand, the large scale and evolving nature of applications in new domains, such as cloud computing, makes the hardware design for applications in such domains even more challenging. Therefore, to make FPGAs a more feasible solution, there needs to be a mechanism to ease the hardware programming, such as high-level synthesis (HLS), which is described below.

## 2.4 High-Level Synthesis

High-level synthesis (HLS) tools try to assist with this issue by raising the level of abstraction and letting the designer use a high-level language such as C/C++ or OpenCL for describing the desired algorithm to generate an RTL design.

HLS tools are becoming increasingly popular due to the recent improvements in their underlying algorithms, which has enabled them to generate RTL designs that have comparable quality with a hand-coded RTL design by an expert hardware designer [7]. High-level synthesis tools use the control/data flow graph (CDFG) of a given program as the main starting point to generate the corresponding RTL de-

sign. Similar to the logic circuit describe in Section 2.2.1, the generated hardware is composed of two main parts: (1) datapath and (2) control path.

The datapath corresponds to the operations and data flow in the the given high-level program while also taking the resource constraints of the target FPGA architecture, such as number of available specific hardware units, into account [7]. The control path is described using an FSM which is constructed after performing two main tasks: (1) scheduling and (2) binding. Scheduling is the process of identifying the cycle in which each operation can be performed given the timing/resource constraints of the target architecture and control and data dependencies in the input application [7]. Binding is the process of mapping the given operations and variables to hardware units that are capable of implementing them while also taking the resource constraints into account [7]. For example, an addition operation is mapped to an adder on the FPGA. If the schedule allows for 20 additions to be performed on the same cycle given the data dependencies, but there are only 10 hardware addition units, the binding task will modify the schedule to perform these operations over two cycles. In a scenario where same hardware unit is shared between multiple operations, the input sources to each operation and the output connections will be defined based on the output of the FSM.

# Chapter 3

# Control-Path Optimization

This chapter studies the potentials of using specialized hardware blocks for implementing Finite State Machines (FSMs) to improve the area efficiency and performance of the control unit portion of the RTL designs generated by high-level synthesis tools. We propose a novel configurable mixed-grained architecture, that makes use of unique characteristics of the FSMs generated by HLS tools to reduce the silicon area that is required for FSM implementation. This is achieved without affecting the control unit performance and, in most of the cases, improves the critical path delay as well.

The rest of the chapter is organized as follows: First we perform analysis on selected characteristics of finite state machines. We show that these characteristics can be used to design an architecture that more efficiently uses the silicon area compared to conventional LUT-based implementation of state machines. We then describe our proposed architecture in detail and a proposed state encoding technique that has been developed in order to better exploit these specialized FSM blocks. Finally we describe the technology mapping algorithm that we have developed to map a given finite state machine to specialized FSM blocks.

## 3.1 Finite State Machine Analysis

In this section we define and analyze specific characteristics of Finite State Machines that can be exploited to design a custom FSM block. We then present the

required state encoding technique to be able to efficiently utilize the custom FSM blocks. Finally, we show that Finite State Machines generated using High-Level Synthesis tools always demonstrate such characteristics, hence they are great candidates to benefit from our proposed customized blocks.

### 3.1.1 Preliminaries

This section presents the preliminaries for our Finite State Machine analysis.

*Definition 1.* **State Transition Diagram**: Finite state machines can be represented by their state transition diagram. State transition diagram is a directed cyclic graph (DCG) $G = (V, E)$ where each vertex $v_i \in V$ represent a unique state and each edge $e_i \in E$ shows a transition between two corresponding states (vertecis). In the rest of this chapter, we refer to vertices and states interchangeably.

*Definition 2.* **Directed Path**: Directed path is a finite sequence of edges following the same direction which connect a sequence of vertices.

*Definition 3.* **Vertex degree**: The degree of a vertex of a graph is defined as the number of edge incidents to the vertex. In DCGs, the vertex degree can be grouped into *fan-in degree* and *fan-out degree* which represent the number of incoming edges and outgoing edges of a vertex respectively.

*Definition 4.* **Branch-Free Path**: Given a DCG, we define a branch-free path to be a directed path where each vertex has at most one fan-out edge but can have more than one fan-in edge. An example of a graph with branch-free paths is shown in Figure 3.1.

### 3.1.2 FSM Characteristic

Using the definitions in the previous section, we now can describe two specific properties of FSMs that can be exploited to reduce the area usage and improve critical path delay of FSM implementations.

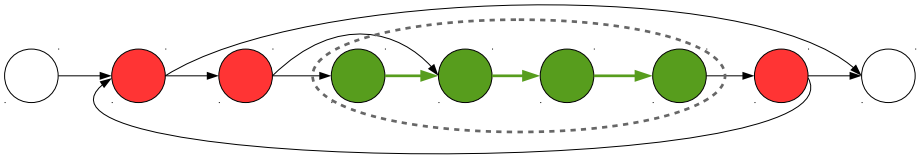


**Figure 3.1:** Branch-free path (shown in green). Each path starts and ends with a vertex with fan-out degree of greater than 1 (shown in red). Note that vertices that belong to a branch-free path can have more than one fan-in edge.

### Abundance of Branch-Free Paths

If the state transition graph of a finite-state machine has long branch-free paths, then consecutive states in each path can be assigned consecutive state values (state encoding) such that next state value can be calculated with a simple increment operation. This leads to a new state encoding where branch-free paths have simple increasing state encoding. This is shown with an example in Figure 3.2. The graph represents part of the state transition diagram of an FSM which contains two branch-free paths labelled with the proposed encoding. Note that the blank states are not part of any branch-free path since they have fan-out degree of 2. Consider the top path with the length equal to $n$, if the first state in this path is assigned the state encoding $X$, then following states in the path will be assigned $X+1$, $X+2$, ... , $X+n-2$, and $X+n-1$ until a non branch-free state is reached. The same rule applies to the second path with the length equal to $m$ where the first state of the path is assigned the state encoding $Y$ and the following states in the path will be assigned $Y+1$, $Y+2$, ... , $Y+n-2$, and $Y+n-1$. Hardware implementation for such state machine has an opportunity to reduce the silicon area, since the next state calculation logic for states that belong to branch-free paths can be realized with a simple adder along with small control logic in hardware. Section 3.2.4 provides more detail on how this adder unit is utilized in our proposed architecture for implementing FSMs.

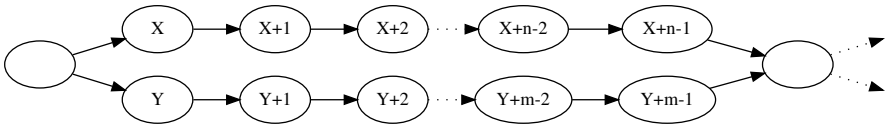



**Figure 3.2:** An example of the state encoding for the states that belong to branch-free paths.

#### Low Fan-Out Degree

For a given FSM, the maximum number of possible next states for any given state can be calculated using the following equation:

$$min(2^q, p)$$

Where $q$ is equal to the total number of inputs to the state machine and $p$ represent the total number of states. However, not all of the input signals are active in different states, therefore the number of reachable states from a given state can be, and often is, far less than the maximum. For each given state, the fan-out degree represent the number of reachable states from that given state.

For the state machines with abundance of branch-free paths, the remaining states which are not part of any branch-free path form a smaller subset of the state machine. If the states that belong this subset have low fan-out degree, there is potential for a hybrid memory-based FSM implementation that is independent of the input size. Section 3.2.4 provides more detail on how a small memory unit is incorporated to our proposed architecture for implementing the next state calculation of the states the do not belong to a branch free path.

### 3.1.3 HLS-Generated Finite State Machines

The results of the analysis on the finite state machines extracted from two sets of HLS benchmarks used in this thesis are presented in Figure 3.6 and Figure 3.7. The details of these benchmark sets are described in Chapter 4. The RTL code for these benchmarks is generated using Vivado HLS, an HLS tool by Xilinx [42].

For the MachSuite benchmarks, we used the default set of HLS optimization directives that were shipped with the benchmarks such as loop unrolling, loop

pipelining, and memory partitioning. In Section 5.6, we specifically analyze the impact of applying HLS optimization directives on generated FSMs by looking at three benchmarks from MachSuite. The HLS directives are obtained using the methodology and results described in [18] which aim to minimize the area-delay product of the generated RTL design. As the results suggest, the size of FSMs and fraction of branch-free paths are not negatively impacted (i.e. the branch-free paths still exist and are a large fraction of total states). In fact, for these three benchmarks, the fraction of branch-free paths actually increases.

For the datacenter benchmarks, BZIP, Lucy, and SQLite (getToken function), HLS optimization directives were applied while generating the RTL design. For the the remaining benchmarks/functions in this benchmark set, no optimization were applied. However, based on our analysis in Section 5.6, we expect a similar behaviour to the result shown for optimizing Machsuite. Applying and evaluating full optimizations on all benchmarks is left to future work.

Figure 3.6 shows that more than 80% of the states in each FSM do not require any input and only have one possible next state, which means they belong to a branch-free path. Figure 3.7, which shows the fan-out-degree (transitions per state) statistics also indicates that there is at most 4 reachable next states for any given state. Therefore, finite state machines coming from HLS-generated RTL codes have a great potential to benefit from our proposed architecture.

### 3.1.4 Data Flow Height Experiment

The FSM analysis on the HLS benchmark sets exposed two common patterns: low fan-out degree and long branch-free paths. This raises the question: what is the cause of the low fan-out degree and long branch-free paths in FSMs among all of the HLS generated RTL codes? Our hypothesis is that these common patterns are caused by data dependent instructions and the latency of instructions within a basic block. To evaluate our hypothesis of data-dependence leading to branch-free paths, we look at the mechanisms used by HLS tools to generate the RTL code for a given application. As discussed in the background section, HLS tools rely on the control/data flow graph (CDFG) of an application and consider the resource and

timing constraints of the target hardware to perform scheduling. The outcome of scheduling is used to generate the control unit and, consequently, the FSM that will direct the operations of the data path.

To understand the impact of data-dependence on scheduling, we mimic the behaviour of an HLS scheduler by constructing a simplified equivalent FSM of a given program from the control flow and data flow graph. Our simplified equivalent FSM assumes that there are infinite resources on the FPGA, the latency of any instruction is one cycle, and that data-dependent instructions cannot take place on the same cycle. These simplifications aim to limit the scheduling to data dependent instructions.

The following steps describe how the simplified equivalent FSM is constructed. Figure 3.3 visualizes this process for the program shown in Figure 3.3a.

- **Step 1:** Construct the control flow graph (CFG) of the program (Figure 3.3a).
- **Step 2**: Construct the data flow graph (DFG) for each of the basic blocks in the CFG. Each node of a DFG shows an operation and edges are representative of data dependencies among these operations (Figure 3.3b).
- **Step 3**: Apply unconstrained list scheduling [21] separately on each of the data flow graphs, with the simplifications described above (Figure 3.3c).
- **Step 4**: Given that each of these data dependent operations may be performed by functional units that require appropriate control signals, each of these operations needs to be a separate state in the equivalent FSM. Replace every cycle of each scheduled DFG with a corresponding state in the equivalent FSM (Figure 3.3d).
- **Step 5**: Finally, connect the states to construct the equivalent FSM. For the states belonging to the same scheduled DFG (within a basic block), apply an edge directly between the states. To construct the transitions between states in different DFGs, replace each control edge between two basic blocks in the CFG with an equivalent edge between states in the FSM. The equivalent edge connects the last state of the predecessor basic block (i.e., cycle N of

the DFG for the predecessor basic block) with the first state in the successor basic block (i.e., cycle 0 of the DFG for the successor basic block)(Figure 3.3d).

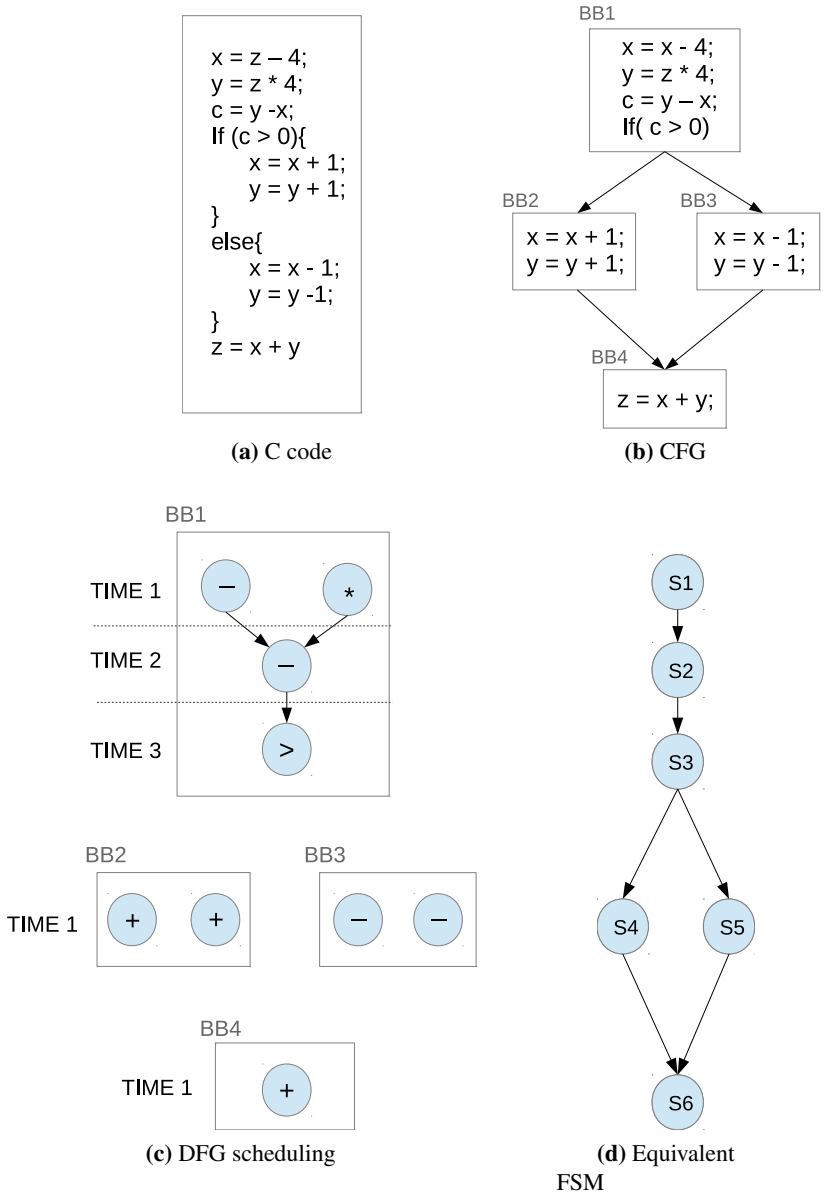
The equivalent FSM constructed by this approach is a naive representation of the FSM that is generated by HLS tools for a given program. For example, multiple operations may be able to be performed on a single cycle, long latency instructions may result in multiple states, or there may be resources limitations in the number of operations that can occur per cycle. However, the simplified FSM maintains the impact of data dependence. We use our simple approach to perform analysis on the equivalent FSMs of SPEC2006 INT benchmarks[32]. Along with the Machsuite HLS benchmarks, we chose the SPEC2006 INT benchmark suite to compare the behaviour of the benchmarks that are and are not necessarily written for HLS.

Figure 3.4a presents the fan-out degree of the equivalent FSM for both the Machsuite and SPEC CPU2006 INT benchmark suites. As can be seen, both benchmarks demonstrate very similar behaviour in the fan-out degree, with over 85% of the states having a single next state. Based on our construction of the equivalent FSM, these single fan-out edges are caused by data dependencies. However, they are independent of input as the timing schedule is predetermined in advance.

Although the simplifications in the equivalent FSM may affect the result of the fan-out degree experiment, the impact will mostly affect the number of states with fan-out degree equal to 1. The result of this experiment (Figure 3.4) shows a very large ratio between single and multi fan-out degrees. Hence, we believe that even with the assumptions discussed above, the equivalent FSM provides a good approximation of the actual FSM to highlight the existence of a large fraction of nodes with a single fan-out edge.

## 3.2 Specialized FSM Block

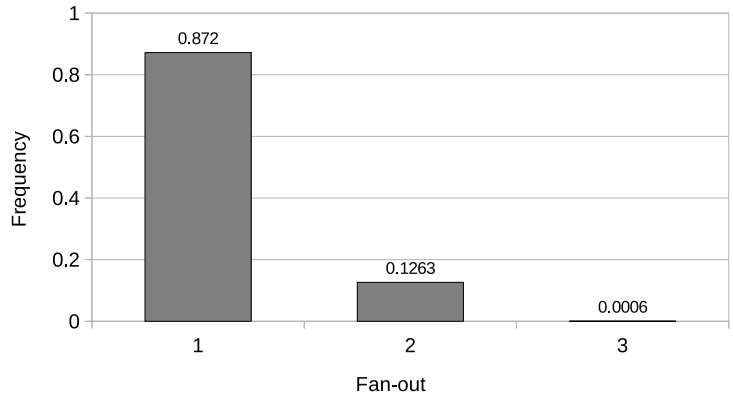
In this section, we first discuss the potential approaches to optimize the implementation of HLS-generated FSMs. We then describe our proposed specialized configurable architecture that takes advantage of the HLS-generated FSM characteristics to implement the FSMs in a more area/delay efficient manner. We also provide



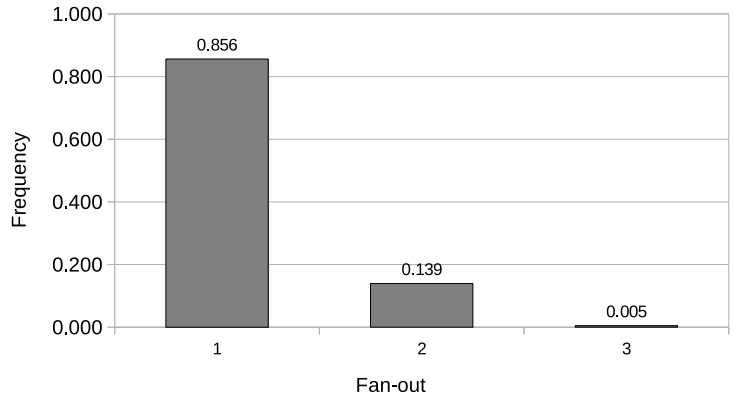

**(a)** C code **(b)** CFG

**(c)** DFG scheduling **(d)** Equivalent FSM

**Figure 3.3:** Equivalent FSM construction




**(a)** MachSuite




**(b)** Spec CPU2006 INT

**Figure 3.4:** Fan-out degree frequency in the equivalent FSMs of MachSuite benchmarks and Spec CPU2006 INT

data that justifies the main design decisions that lead to the final architecture of this hybrid architecture.

### 3.2.1 Design Space Exploration

There are different approaches to exploit the properties of HLS-generated FSMs to optimize the FSM implementation on FPGAs. These approaches along with their benefits and drawbacks are described below:

### Specialized Hard Blocks

One potential solution is to introduce a specialized hard block to FPGA Architecture that is designed to only implement the HLS-generated FSMs. Such specialized hard block is extremely efficient compared to the FPGA LUT-based logic blocks due to the reduced overhead that comes with the flexibility of LUT-based logic blocks. In this thesis, we propose and evaluate a novel mix-grained architecture that makes use of such hard blocks to improve the efficiency of FSM implementation on FPGAs.

### Soft-Logic and Block RAMs

Another potential approach is to use FPGA soft-logic to implement an adder unit that interacts with the existing embedded memories on FPGAs to implement the FSM in a more efficient manner. In this approach, the synthesis tool is responsible to transform the FSM description to a logic circuit that implements the FSM using an adder, control logic, and memory.

This approach does not require any modification to the existing FPGA architecture. However, due to the overhead of the soft-logic implementation of the adder and control logic along with using programmable routing between the adder, control logic, and memory unit, this solution is not as efficient as having specialized hard blocks. Evaluation of this approach is left to future work.

### Modified Block RAMs

Another solutions is to modify the existing block RAMs on FPGAs by adding a hard adder and control logic to these blocks such that they can support our proposed FSM implementation. This is a promising approach, however, the existing block RAMs on FPGAs are typically synchronous memories. Therefore, digital designs that are not latency tolerant are not able to be mapped to such block RAMs, since a synchronous memory read adds one cycle delay to the state calculation process. Evaluation of this approach is left to future work.

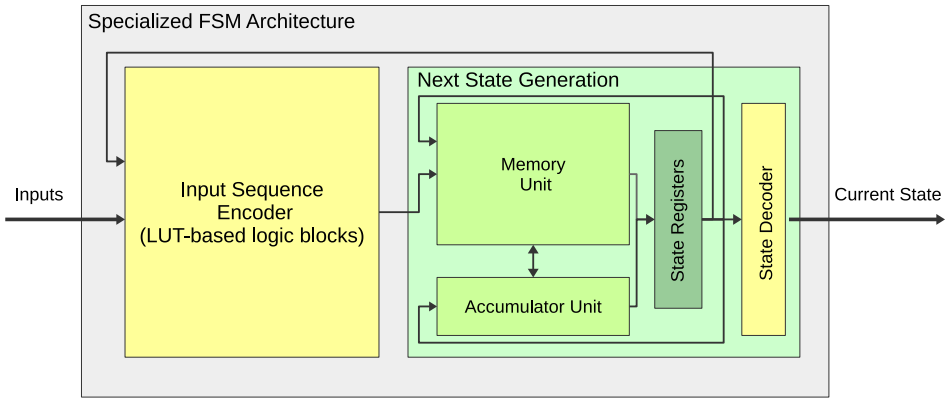



**Figure 3.5:** High-level view of the specialized FSM architecture

### 3.2.2 Mix-Grained Architecture

Our proposed architecture consists of both fine-grained (soft) and coarse-grained (hard) logic that are connected via hard (or flexible FPGA) routing, which together form the mix-grained architecture. The high-level architecture is illustrated in Figure 3.5. The coarse-grained part of this architecture implements the next state calculation and consists of two main units, the accumulator unit and memory unit. The accumulator unit takes care of state calculation for the states that belong to the branch-free paths, while the memory unit stores the next states for the remaining states, along with some metadata that will be described later. As previously mentioned, these remaining states tend to have a low fan-out degree, which makes them well suited to be stored in memory, since fan-out degree directly corresponds to the number of entries per-state in memory. The fine-grained part of this architecture takes the current state and input signals and tries to minimize the address space of the memory unit, and hence, the memory size. As mentioned in Section 3.1.3, the reduction of the state address space is possible since the number of reachable states from a given state is often much less than the maximum possible number of reachable states.

The following sections describe the the coarse-grained and fine-grained parts of the proposed architecture in more detail.

### 3.2.3 Input Sequence Encoder Unit

The input sequence encoder unit implements a configurable encoder using the FPGA soft logic. Figure 3.6 and Figure 3.7 explain the intuition behind having such unit. At each state of the state machine only a subset of input signals impact the state transition. This subset of inputs are called *active inputs*. FSMs extracted from our benchmarks sets have variable number of inputs ranging from 3 to 56, however, the number of active inputs at each state is much less for these benchmarks (characteristics of FSMs are listed in detail in Chapter 4). As shown in Figure 3.6, the number of state machine active inputs per state varies from 0 to 5, however, the number of next reachable states from a given state (i.e number of fan-outs per node in the state transition graph) does not exceed 4. This means that the choice of next state, which corresponds to the memory address, can be represented by only 2 bits instead of 56. Therefore, we use a simple encoder that maps the possible large input sequence for the state machine to a smaller sequence of length $\log_2$ (maximum number of reachable states per state). This significantly reduces the size of the memory unit that is used for next state calculation as it enables us to avoid storing don't care data for unreachable states. The input sequence encoder unit can be easily implemented on a LUT-based cluster as part of the conventional FPGA architecture.

### 3.2.4 Coarse-Grained Fabric

The coarse-grained fabric corresponds to the "Next State Generation" block in Figure 3.5. By analyzing the edge distribution of the state transition graphs among our benchmark suite (discussed more in Chapter 4), we observed abundance of branch-free paths, states with only one next state where the transition between states is input-independent. In Section 3.4 we describe a simple encoding that enables using a single accumulator in order to calculate the next state value for such states.

Figure 3.8 presents a detailed breakdown of the next state generation block shown in Figure 3.5. The following subsections describe each of the components in more detail.

There are timing requirements for the FSM block that require delay of certain

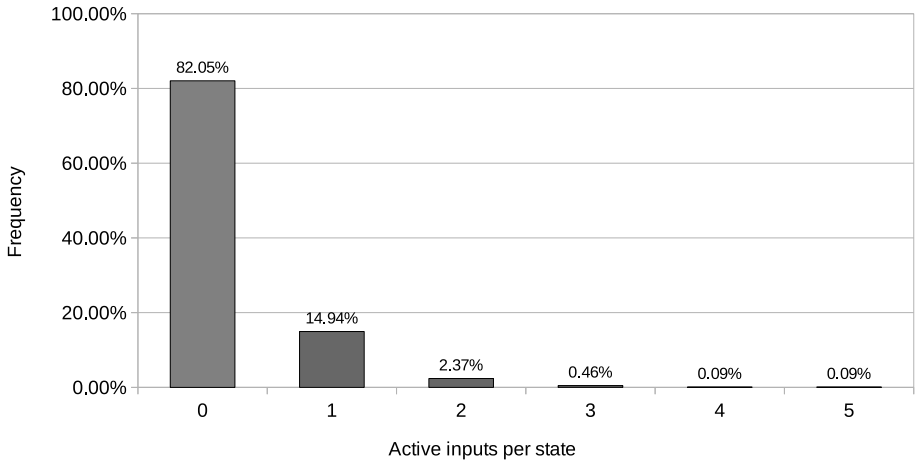



**Figure 3.6:** Number of active inputs per state calculated as average over 46 FSMs extracted from 21 benchmarks generated by HLS. Details of the benchmarks are described in Chapter 4.

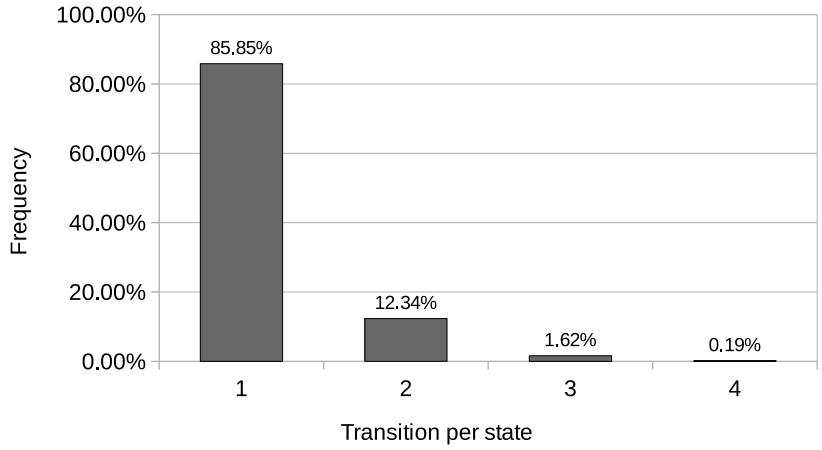



**Figure 3.7:** Edge distribution: number of transitions per state calculated as an average over 46 FSMs extracted from 21 benchmarks generated by HLS. Details of the benchmarks are described in Chapter 4.

data. For example, metadata read from an entry in memory corresponds to the next state and in case of the branch-free paths, metadata is used for the entire path as

such we require a mechanism to save the metadata. That is why we use the registers to delay the metadata by one cycle such that they apply to the next state and in the case of the branch-free path to the $l$ next following state where $l$ is the length of the path. ”Path Final State Register”, ”Branch Address Register”, and ”State Control Register” are the registers that we have used for this purpose which are explained in detail below.

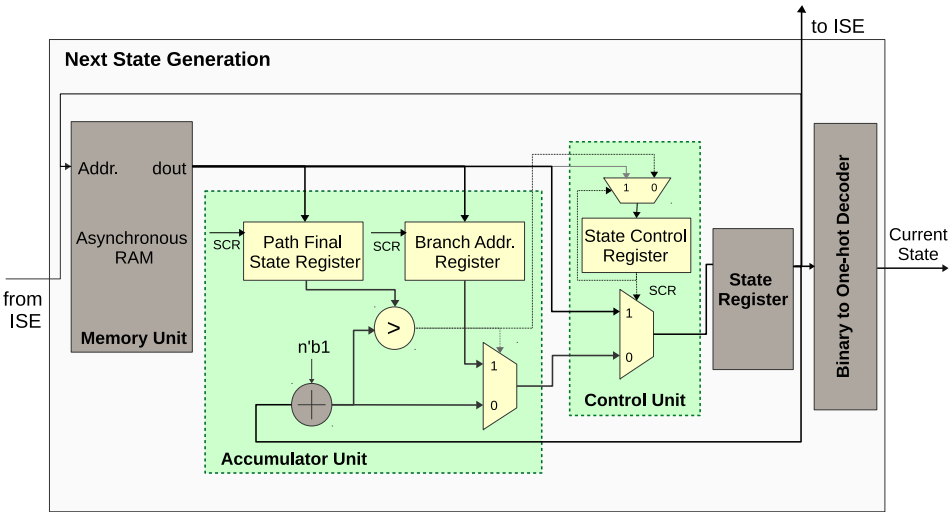



**Figure 3.8:** Next state generation block

### Accumulator Unit

This unit is responsible for calculating the next state values for the trailing input-independent states on a branch-free path. After applying the proposed state encoding, the next state value for states that belong to a given branch-free path in a state transition graph can be calculated using a simple accumulator along with some extra information, which are described below:

- **Adder**: The adder is the main part of the accumulator unit. It takes in the current state and increments the value to calculate the next state in the branch-free path. It has two inputs: a single 1-bit value set to one, and the current state value coming from the output of the state register.

- **Control logic**: While an adder is enough to calculate the next state value for the branch-free states, it is not sufficient to determine when we have reached the end of the branch-free path. Additionally, once we have reached the end of the branch-free path, we need to read the next state value from memory. However, the address to this state is not just the current state encoding, since the memory address space is separate from the state encoding for branch-free paths. Therefore, we use two metadata registers to store this information for each path.

  - The **Path Final State Register** is set to the state value of the last state on the path. This is used to mark the ending state of the path.
  - The **Branch Address Register** is set to the address we should read from the memory once we have reached the end of the branch-free path.
  - The **comparator** is used to compare the values of the path final state register with the output of the accumulator, and then generates the control signal which decides if the next state value should come from the accumulator or the Branch Address Register.

#### Memory Unit

The memory unit is responsible for storing the next state value for the sates that do not belong to a branch-free path along with some metadata to assist with the transition from the accumulator unit to memory unit. To avoid adding an extra cycle delay to the next state calculation, an asynchronous memory block must be used for this unit. Figure 3.9 shows the content of a row in memory. It consist of four main fields: (1) Next State Value (2), Path Final State, (3) Branch Address, and (4) State Control bit. The first and fourth fields always have a valid value, however the second and third fields will only be valid in the case where next state belong to a branch-free path. In this case, the contents of these two fields will be registered into the registers described in the accumulator unit, as described above. The last field, state control bit, is used to determine if the source of the next state value should be the accumulator unit or memory unit. This field will be registered into the control unit register that will be described below.

The depth of the memory is dependent on the non-branch-free states and edge distribution and the width is based on the next state plus metadata. In Chapter 5 we describe how we size the memory unit in detail.


**Figure 3.9:** Memory content

### Control Unit

The control unit is responsible for selecting the source of the next state value between the accumulator unit and memory unit using a multiplexer which is controlled by the "State Control Register". The State Control Register can be set in two different ways: (1) The State Control Field of the memory unit for the given state Figure 3.9, or (2) the result of the comparator in the accumulator unit which marks the termination of the branch-free path, Figure 3.8). At any given time, either the memory unit or the accumulator unit is active and responsible for calculating the next state value. The active unit is responsible for selecting whether the same unit is active on the next cycle or the other unit is active on the next cycle. This is implemented as a feedback loop from the State Control Register to the select signal of the multiplexer feeding the State Control Register. This continues to select the same unit until that unit signals a transition by toggling zero to one or vice versa.

### State Decoder

We provide an optional binary to one-hot decoder at the output of this block to enable more efficient binary to one-hot conversion if required by the rest of the circuit.

## 3.3 Fracturable FSM Hard Blocks

The size of finite state machines can vary significantly among different applications. As mentioned in Chapter 2, any hard block on FPGAs will be wasted if not fully utilized by the applications, leading to fragmentation in the hard blocks. Therefore, to be able to efficiently accommodate state machines with various number of states, we propose fracturable FSM hard blocks. The main idea behind having fracturable FSM blocks is to tailor the block size such that it accommodate the state machines with an average size while supporting combination of two blocks such that they can accommodate large FSMs that do not fit into just one block.To map a large state machine to multiple smaller combined blocks, the state machine needs to be partitioned to multiple sub state machines, and the architecture should enable fast transition between these blocks. In this work we only look at partitioning state machines to two sub-machine which enables us to accommodate all the FSM benchmarks that we use, however, our approach can be easily applied to more partitions. We first describe our partitioning method and then propose minor modifications to the FSM hard block to support FSM block combination.

### 3.3.1 FSM Partitioning

Given an input graph $G = (V, E)$, the objective of bi-partitioning problem is to partition the vertex set $V$ into two disjoint subsets with the main goal of minimizing the number of edges between two subsets. The FSM partitioning (decomposition) problem is a famous problem [21] with plenty of possible solutions which are mainly proposed to target complex state machines. For the purpose of partitioning HLS-generated FSMs, which are less complex in terms of the number of transition between different states, we chose a classic algorithm known as Fiduccia-Matheyses partitioning algorithm [9] and, with an example in Chapter 5, show that it works very well for the state machines that are generated by HLS tools. Fiduccia-Matheyses partitioning algorithm is an iterative mincut heuristic algorithm with a linear computation time with respect to the size of the input graph.

### 3.3.2 Fracturable FSM Block Architecture

When splitting an FSM over multiple fracturable blocks, every state transition across two different blocks requires control signals that enable switching between these blocks. For example, if the current state *X* is mapped to the fracturable block A and the next state *Y* is mapped to the fracturable block B, then when the transition occurs, the state register of block A must enter the idle state (described later), and the state register of block B must be updated to *Y*. To enable this switching between the blocks, state *X* must carry the metadata that controls this transition. A potential candidate to store this metadata is the memory unit. If the states that mark the transition across fracturable blocks are stored in memory, an extra field on each memory row can be used to store the required metadata for transitioning across the blocks. In this example, state *X* must be stored in memory. For this work, we only allow splitting the FSM over two fracturable blocks, thus a single bit in memory is sufficient to indicate whether the next state should be calculated in this block or the other fracturable block that implements the same FSM

By transitioning to another fracturable block, we enter a new state which can either be mapped to the memory unit or accumulator unit. As described in Section 3.2.4, if this state is mapped to the accumulator unit the control registers, specifically, Path Final State Register, Branch Address Register, and State control Register must be updated as well.This requires extra multiplexer logic that allows setting the value of these registers from multiple sources which are the memory unit on the same block as well as the memory unit on the other fracturable block. To simplify this scenario and reduce the overhead logic, we decide to only map this state to the memory unit to avoid the need for updating the control registers described above.

To summarize, for any transition across two fracturable blocks, both current state and next state must be stored in memory. Although this increases the required memory size to accommodate the FSM, a proper partitioning algorithm that aims to reduce the number of transitions between blocks can limit this memory overhead to only a few extra entries.

Additionally, we must add a multiplexer before the state register in each fracturable block to allow updating the state value using the data stored in the other

fracturable block, for the scenario where there is a transition between two blocks. We dedicate the state value *zero* as the idle state. Once the state calculation tasks gets transferred over to the other fracturable block, the inactive block will enter the idle state by updating its state register to *zero*.

The overheads and issues of splitting the FSM over more than two fracturable blocks are discussed in Chapter 5.

## 3.4 State Assignment Algorithm

The state assignment (encoding) problem is defined as determining the binary representation of the states in a finite-state machine such that each state has a unique value to separate it from the other states [21]. The state encoding directly affects the circuit area and performance as different encoding results in different circuit complexity. The choice of circuit implementation, such as two-level logic, multiple-level logic, or in our case a mix-grained architecture that contains a specialized ASIC-like FSM block, also plays an important role in finding the state encoding that optimizes the circuit area and/or performance. For the purpose of mapping to our specialized FSM architecture, the circuit area is measured by the number of bits required for state encoding, number of states that have to be mapped to the memory unit, and the logic complexity of the input sequence encoder. We propose a novel state assignment technique for the FSM targeting our FSM block. This technique aims to minimizes the FSM area by mapping as many states to the accumulator logic as possible and minimizing the number of states that reside in memory, hence reducing the input encoder logic complexity.

Our state assignment algorithm consists of two main parts: (1) identifying the state categories and (2) performing state encoding separately on each state category. Before describing how we categorize the states, we first explain why we need different categories. Our proposed FSM block contains two main parts that perform the next state calculation: the memory unit and accumulator unit. Each units is responsible for determining the next state value for a subset of states. A proper state encoding for each subset must consider the limitations and requirements of the unit that is in charge of the next state calculation for this subset. This lead us to group the states into two categories based on whether their next state

is calculated by the memory unit or accumulator unit. Below, we discuss the requirements of each unit in detail and explain the actions required to meet these requirements. Then we explain how to categorizes the states.

### Memory Unit Requirements

The main requirements are the memory size and address signal generation. The read address signal of the memory unit is formed by concatenating the value of the current state and encoded input signals that comes from the input sequence encoder. However, only a subset of states of the FSM reside on the memory, hence not all the bits of the current state signal are necessary for addressing the memory. For example, if the number of the states that are stored on memory is equal to $n$, then only $\log_2 n$ bits of the current state signal are required for addressing the memory. Therefore, the state encoding for these states must be between *zero* and $n$ to minimize the size of memory.

### Accumulator Unit Requirements

As described in the Section 3.2.4, the accumulator unit performs the state calculation for the states that belong to branch-free paths, hence it is necessary for the consecutive states of each path to have consecutive state values. However, there must be one and only one state encoding for each individual state, therefore in a scenario where two branch free paths overlap, such as path A and path C shown in Figure 3.10b and Figure 3.10c, we must first refine the paths such that they do not overlap to avoid two encoding values for the same state.

### Path Refinement

As we discussed in the previous section, on any transition from a memory state to a counter state, there is metadata for the corresponding branch-free path that must be provided to the accumulator unit. To store this metadata, we use the memory location for a state that transitions to a branch-free state. For any given path, the previous state that branches to this path must reside in memory so it can store this metadata. As such, there must be a gap of at least one memory state between the

vertices of any two branch-free paths. Note that due to the definition of a branch-free path, any two non-overlapping branch-free paths satisfy this requirement, since a branch-free path begins right after, and terminates where, there is a state with a fan-out degree greater than 1 (divergent vertex), which corresponds to a state stored in memory. Thus, any two non-overlapping paths will be at least one memory state away from each other.

Two branch-free paths can never overlap at the starting vertex since they will be equivalent. However, they can overlap on any other vertex, in which case the remaining vertices will also overlap. Therefore, if two branch-free paths overlap on any of their vertices, they will definitely overlap on the ending vertex as well. The ending vertex is always a predecessor to a divergent vertex. This means that branch-free paths that have different starting vertices but share a common termination divergent vertex, might partially overlap with each other. We use this condition to find the potentially overlapping paths by grouping the paths that share a common termination divergent vertex. In a scenario where the branch-free paths overlap, we must refine the paths such that the refined paths are at least one memory state away from each other as described above.

The pseudo code of our proposed path refinement algorithm is shown in Algorithm 1. The input to the algorithm is a set of branch-free paths which share a common termination vertex. This means that the ending node of all paths in this set is a predecessor to a common divergent vertex (a vertex with fan-out degree greater than one). Note that due to the definition of a branch-free path, paths that do not share a common termination node will never overlap, hence this is a required condition that indicates the potential of overlapping. After applying the path refinement algorithm, the output is (1) a set of refined branch-free paths and (2) a set of *independent* vertices which, contains the states that initially belong to overlapping branch-free paths, but are no longer part of the refined paths after applying refinement. The path refinement algorithm will be used as an intermediate step by our state assignment algorithm, Algorithm 2, which will be described after the path refinement algorithm. We start by describing the details of the path refinement algorithm and then use an example to better illustrate these steps.

- **Step 1**: At the initial step, set $\mathbb{S}$, which will eventually contain all vertices

that belong to the refined branch-free paths is empty. This set is used to keep track of the vertices that belong to the refined branch-free paths over different iterations of this algorithm to help detect the overlaps. Second, we sort the paths that belong to the input set $\mathbb{G}$ based on their path length and then add them to *SPL*, a sorted list of all paths from $\mathbb{G}$ (lines 3-4).

- **Step 2**: At this step, as long as *SPL* is not empty, we select the longest path *LP* from *SPL* to apply the path refinement process on it (lines 5-6).

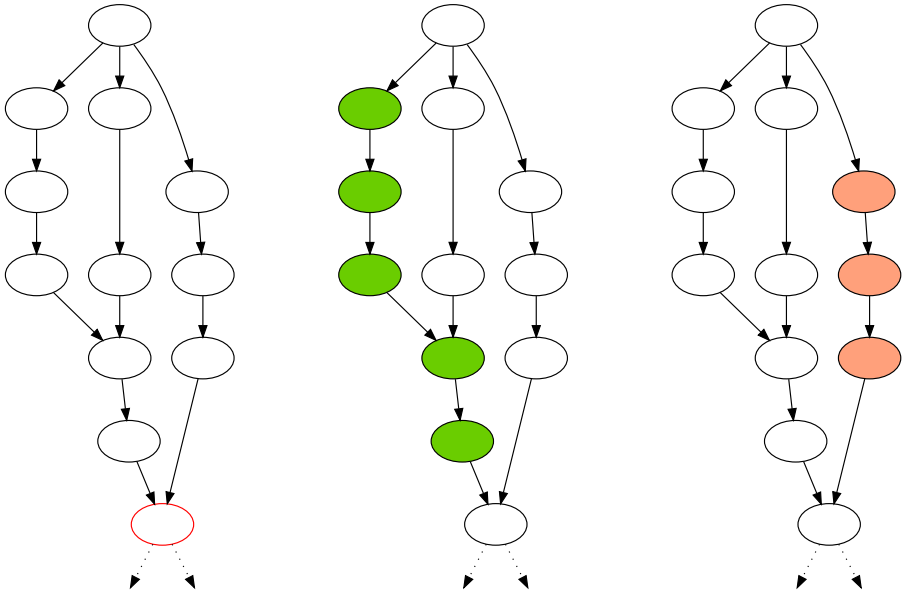
- **Step 3**: Next we traverse *LP* and compare each of its vertices with every vertex of $\mathbb{S}$, until we find a common vertex between *LP* and $\mathbb{S}$ or we reach the end of path *LP* (lines 7-13). Note that when we first start the algorithm, set $\mathbb{S}$ is empty, thus none of the vertices of *LP* will overlap with the vertices of set $\mathbb{S}$ for the first path.

- **Step 4**: After we detect two overlapping vertices at $v_i$, we must terminate LP at $v_{i-1}$. This requires cutting *LP* such that the predecessor of $v_{i-1}$, $v_{i-2}$, is the ending vertex of the refined path. By doing so, $LP_{refined}$ no longer overlaps with any of the paths that have already been refined. Vertex $v_{i-1}$ will now become an independent state and be added to the set of independent states $\mathbb{I}_G$ which will be stored in memory. This independent state, $v_{i-1}$, separates $LP_{refined}$ from all others refined paths (lines 9-10).

- **Step 5**: Next, we add the refined path, $LP_{refined}$, to the set of refined paths $\mathbb{G}_{refined}$ and add all of its vertices to the set of refined path vertices $\mathbb{S}$ (lines 15-16) to be used for the next iterations of the while loop (line 5).

- **Step 6**: Once the while loop is completed, $\mathbb{G}_{refined}$ will contain the set of refined branch-free paths and $\mathbb{I}_G$ will include the independent states.

An example of a scenario when two branch-free paths of a state transition graph overlap is illustrated in Figure 3.10. Figure 3.10a shows part of the state transition graph of an FSM (labels are omitted for simplicity), which contains 3 branch-free paths. Figure 3.10b, Figure 3.10c, and Figure 3.10d highlight these paths individually. These three paths all share a common termination node (show in red in Figure 3.10a), thus they might overlap. In this case, the last two states of path




**(a)** state diagram **(b)** path A **(c)** path B

**(d)** path C **(e)** refined paths

**Figure 3.10:** Path refinement

**Algorithm 1** Path Refinement

**Input:** $\mathbb{G} \rightarrow$ Set of branch-free paths grouped by common termination node
**Output:** $\mathbb{G}_{refined} \rightarrow$ Set of refined branch-free paths from $\mathbb{G}$
**Output:** $\mathbb{I}_G \rightarrow$ Set of independent vertices from $\mathbb{G}$
1: $\mathbb{S} \rightarrow$ Set of refined path vertices
2: $SPL \rightarrow$ sorted list of branch-free paths

3: $\mathbb{S} = \varnothing$
4: $SPL$ = sort($\mathbb{G}$);
5: **while** $SPL! = \varnothing$ **do**
6: $LP$ = select the path with the longest length from $SPL$;
7: **for all** $v_i \in LP$ **do**
8: **if** $v_i \in \mathbb{S}$ **then**
9: $LP_{refined} = terminate(LP, i-1)$;
10: add $v_{i-1}$ to $\mathbb{I}_G$;
11: break;
12: **end if**
13: **end for**
14: **if** $LP_{refined}! = \varnothing$ **then**
15: add $LP_{refined}$ to $\mathbb{G}_{refined}$
16: add all the vertices of $LP_{refined}$ to $\mathbb{S}$
17: **end if**
18: **end while**

A and C overlap, therefore the path refinement algorithm must be applied on these paths. An example used to illustrate this algorithm is described below:

In this example, the input to the algorithm is a set of branch-free paths that contains path A, path B, and path C which all share a common termination node (show in red in Figure 3.10a). At step 1, these paths are sorted based on their path length in the following order, path A (5 vertices), path C (4 vertices), and path B (3 vertices). At the first iteration of the algorithm, path A will be selected. However, as the set of refined path vertices $\mathbb{S}$ is empty, path A do not require refinement and will be added to the set of refined paths $\mathbb{G}_{refined}$ as is. All of its vertices will then be added to the set of refined path vertices $\mathbb{S}$ (step 2 through 5). At the second iteration, path C will be selected. By comparing each of its vertices

with the vertices of set $\mathbb{S}$, which now contains all the vertices of path A, we find that the third vertex of path C already exist in set $\mathbb{S}$. Therefore, we terminate path C at its second vertex by cutting it after its first vertex. This means that one independent memory state, the second vertex of path C, will be used before overlapping with path A to store the necessary metadata to join path A. Figure 3.10e illustrates the effect of the terminate subroutine applied to path C from Figure 3.10d. After applying terminate to the middle path, the refined path C now only has one state. The one state gap that separates path A from the refined path C is labelled "indep." (independent) state. At the third iteration of the algorithm path B, the only remaining path, will be selected. Since non of the vertices of path B overlap with any of the refined paths, it will be added to the set of reined paths $\mathbb{G}_{refined}$ as is. At this point the algorithm is completed and the output is (1) the set of refined paths A, B, and C (shown in Figure 3.10e), and (2) a set of independent states which contains the vertex labelled "indep.".

**State Assignment**

Next, we will describe the full state assignment algorithm, which is presented in Algorithm 2. As mentioned in the beginning of this section, the state assignment algorithm consists of two main parts: (1) identifying the state categories and (2) performing state encoding separately on each state category. These state categories are described below:

- **branch-free states**: States that belong to non-overlapping branch-free paths.
- **independent states**: All remaining states that either have a fan-out degree greater than one (divergent states), or states that are initially part of the overlapping branch-free paths but do not qualify to remain part of the path after applying path refinement.

Below we describe the details of the state assignment algorithm (Algorithm 2):

- **Step 1 (Identify divergent vertices)**: Identify and add vertices with a fan-out degree greater than one (two or more successors) to the set of divergent vertices $\mathbb{D}$ (lines 9-10).

**Algorithm 2** State Assignment

**Input:** $G_{fsm} = (V,E) \rightarrow$ FSM state transition graph

**Output:** $G_{encoded-fsm} = (V,E) \rightarrow$ Encoded FSM state transition graph where each vertex is labelled by its state encoding value

1: $\mathbb{P}_{non-refined} \rightarrow$ Set of non-refined branch-free paths
2: $\mathbb{P}_{refined} \rightarrow$ Set of refined branch-free paths
3: $\mathbb{P}_{k-non-refined} \rightarrow$ Set of non-refined branch-free paths that share common terminating divergent vertex $d_k$ ($\mathbb{P}_{k-non-refined} \subset \mathbb{P}_{non-refined}$)
4: $\mathbb{P}_{k-refined} \rightarrow$ Set of refined branch-free paths after applying refinement algorithm on $\mathbb{P}_{k-non-refined}$ ($\mathbb{P}_{k-refined} \subset \mathbb{P}_{refined}$)
5: $\mathbb{I} \rightarrow$ Set of independent vertices
6: $\mathbb{I}_k \rightarrow$ Set of independent vertices found after applying path refinement algorithm on $\mathbb{P}_{k-non-refined}$
7: $\mathbb{D} \rightarrow$ Set of divergent vertices
8: $\mathbb{S}_i \rightarrow$ Set of successors of divergent vertex $d_i$

9: /*find all divergent vertices in the state transition graph*/
10: traverse $G_{fsm}$ and populate $\mathbb{D}$ with the vertices that have fan-out greater than 1
11: /*find all branch free paths in the state transition graph*/
12: **for all** $d_i \in \mathbb{D}$ **do**
13:  **for all** $s_j \in \mathbb{S}_i$ **do**
14:  add the branch-free path $p_j$ that starts from $s_j$ to $\mathbb{P}_{non-refined}$
15:  **end for**
16: **end for**
17: /*group together the branch-free paths that share common terminating divergent vertex*/
18: **for all** $d_k \in \mathbb{D}$ **do**
19:  add every branch-free path from $\mathbb{P}_{non-refined}$ that share common terminating vertex $d_k$ to $\mathbb{P}_{k-non-refined}$
20: **end for**
21: /*Apply the path refinement algorithm (Algorithm 1)*/
22: **for all** $d_k \in \mathbb{D}$ **do**
23:  ($\mathbb{P}_{k-refined}$, $\mathbb{I}_k$) = path refinement ($\mathbb{P}_{k-non-refined}$) /* Algorithm 1*/
24:  $\mathbb{P}_{refined} = \mathbb{P}_{refined} \cup \mathbb{P}_{k-refined}$
25:  $\mathbb{I} = \mathbb{I} \cup \mathbb{I}_k$
26: **end for**
27: $\mathbb{I} = \mathbb{I} \cup \mathbb{D}$

28: /*state assignment*/
29: **for all** $v_i \in \mathbb{I}$ **do**
30: assign a state encoding in an incrementing manner starting from zero
31: **end for**
32: **for all** $P_i \in \mathbb{P}_{refined}$ **do**
33: **for all** $v_j \in P_i$ **do**
34: assign a state encoding in an incrementing manner starting from the last value that was used for the previous path +1
35: **end for**
36: **end for**

- **Step 2 (Identify branch-free paths)**: Find all of the branch-free paths between every two divergent vertices that have been marked in the first step and add them to the set of non-refined branch-free paths $\mathbb{P}_{non-refined}$ (lines 11-16). To identify a branch-free path, we start from a successor of a divergent vertex and add its consecutive vertices to the path by traversing the graph before arriving at another divergent vertex. By doing so, all the vertices on this path will only have a fan-out degree of one, hence the corresponding path meets the requirements of a branch-free path.

- **Step 3 (Group the paths based on their termination vertex)**: At this step, the branch-free paths that share a common termination divergent vertex $d_k$ will be grouped together and added to $\mathbb{P}_{k-non-refined}$, since this is a precondition for potential overlapping paths (lines 17-20).

- **Step 4 (Apply path refinement)**: Apply the path refinement algorithm (Algorithm 1) on each group of branch-free paths with a common termination vertex that were obtained in step 3, $\mathbb{P}_{k-non-refined}$ (line 23). The output of this step is the subset of refined branch-free paths, $\mathbb{P}_{k-refined}$, and the subset of independent states, $\mathbb{I}_k$, that are no longer part of the refined paths (described in detail in Algorithm 1).

- **Step 5 (Update state categories-1)**: Add the subset of paths that were refined in step 4, $\mathbb{P}_{k-refined}$, to the final set of refined branch-free paths, $\mathbb{P}_{refined}$ (line 24). Update the set of independent vertices $\mathbb{I}$ by adding the vertices that were obtained in step 4, $\mathbb{I}_k$, to this set (line 25).

- **Step 6 (Update state categories-2)**: Add the divergent vertices, $\mathbb{D}$, to the list of independent vertices $\mathbb{I}$. Set $\mathbb{I}$ indicates all of the vertices(states) that will be mapped to the memory unit (line 27).

- **Step 7 (State assignment-1)**: Finally, for the independent vertices, $\mathbb{I}$, that were identified in step 1 through step 6, assign incremental values to the vertices(states) starting from zero (lines 29-31).

- **Step 8 (State assignment-2)**: For each branch-free path in the refined path set $\mathbb{P}_{refined}$ , assign incremental values to the consecutive vertices (states). For the first path, the starting state value will be the value assigned to the last independent state (step 7) plus one. For all remaining paths, the starting state value is one greater than the last state value of the previous path (lines 32-36).

## 3.5 Mapping to the Specialized FSM Architecture

In this section we describe the steps required to map a given state machine to our specialized FSM architecture. The mapping process consists of 3 main steps which are listed below:

- Applying the size checking pass
- Fine-grained mapping
- Coarse-grained mapping

### 3.5.1 Applying the Size Checking Pass

At this step we check two required conditions to verify whether the input FSM, described by its state transition table, is suitable to be mapped to the next state generation block: (1) Whether the number of bits required for the encoding of the state machine is smaller than the maximum bit-width of the adder unit, (2) if the total number of states that reside in memory are smaller than the size of the memory unit. This step is performed after applying the state assignment algorithm

described in Algorithm 2. A more detailed description of this step is described in the Chapter 5.

### 3.5.2 Fine-Grained Mapping

This part corresponds to mapping the corresponding part of the FSM to the input sequence encoder. To do so, we must form the logic function that implements the input sequence encoder.

This is achieved by performing a transformation on the state transition table. The goal of this transformation is to reduce the number of inputs to what we call the *encoded input sequence*. This is shown with an example in Table 3.1. Table 3.1a shows the original state transition table of an example FSM. This table shows the choice of next state value for a given state based on the current state and input value. This FSM has 10 inputs, however, each state has no more than 3 next states. Therefor, an *encoded input sequence* with only 2 bits is sufficient to distinguish among the next states of any given current state. To obtain such *encoded input sequence*, we first transform the original state transitions table to a reduced table which only contains the states that have more than one next state. The reduce table is shown in Table 3.1b. We then use the reduced table as a truth table to implement a logic function that takes the *state machine inputs* as input and generates an *encoded input sequence* as output.

### 3.5.3 Coarse-Grained Mapping

The next step, coarse-grained mapping, generates the memory contents for the FSM. At this point, the state assignment algorithm (Algorithm 2) has been applied to the state machine. Hence the states that reside in memory, and their corresponding metadata have been determined. Using this information, the memory contents are generated in the format shown in Figure 3.9.

## 3.6 Putting it All Together

In this section, we present a complete example from C code to implementation and operation of the corresponding FSM on our proposed specialized FSM block.

| Input | Current state (cs) | Next state (ns) |
|---|---|---|
| 10'bx | s0 | s1 |
| 10'bx | s1 | s2 |
| 10'bx | s2 | s3 |
| 10'bxxx11xxx0x | s3 | s17 |
| 10'bxxx11xxx1x | | s4 |
| 10'bx | s4 | s5 |
| 10'bx | s5 | s6 |
| 10'bx | s6 | s7 |
| 10'bx | s7 | s8 |
| 10'bx | s8 | s9 |
| 10'bx | s9 | s10 |
| 10'bx11xxx1xxx | s10 | s11 |
| 10'b101xxx11xx | | s3 |
| 10'bx11xxx1xx0 | | s17 |
| 10'bx | s11 | s12 |
| 10'bx | s12 | s13 |
| 10'bx | s13 | s14 |
| 10'bx | s14 | s15 |
| 10'bx | s15 | s16 |
| 10'bx | s16 | s17 |
| 10'bx | s17 | s0 |

**(a)** Original state transition table

| Original input | Current state (cs) | Encoded input |
|---|---|---|
| 10'bxxx11xxx0x | s3 | 2b00 |
| 10'bxxx11xxx1x | | 2b01 |
| 10'bx11xxx1xxx | s10 | 2b00 |
| 10'b101xxx11xx | | 2b01 |
| 10'bx11xxx1xx0 | | 2b10 |

**(b)** Truth table for the Input Sequence Encoder

**Table 3.1:** Input sequence encoder generation: the original transition table is given in (a), the reduced table is given in (b). {cs, active input(2bits)} will be used to address the memory instead of {cs, inputs(10 bits)}.

Additionally, this example highlights different sources of branch-free paths in HLS generated benchmarks.

```
1.  int foo(int A, int B, int C, int N)
2.  {
3.      int x, y, z, result;
4.
5.      x = y = z = result = 0;
6.      if (B > A){
7.          int tmp = B;
8.          B = A;
9.          A = tmp;
10.     }
11.     while( (N-- > 0) && (0 < A) ) {
12.         x = A / B;
13.         A = x;
14.         y = x << C;
15.         z = y + x;
16.         result += z;
17.     }
18.
19.    return result;
20.  }
```

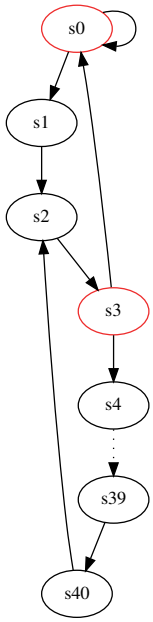
**Figure 3.11:** An example C code: the example shows a while loop with four consecutive instructions each with a data dependency on the previous instruction. Additionally each instruction performs an operation that has a different latency in hardware such as division, multiply, shift, and add.

### 3.6.1 Generating the FSM

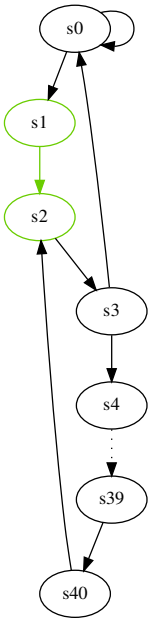
Figure 3.11 presents a simple microbenchmark with a function, *foo*, that contains conditional code (line 6), a loop (line 11), data-dependent instructions (lines 7-9 and 12-16), and instructions with different latencies (e.g., divide on line 12 and shift on line 14).

Figure 3.12a presents the FSM generated by Vivado HLS as part of the generated RTL design for the C function shown in Figure 3.11.

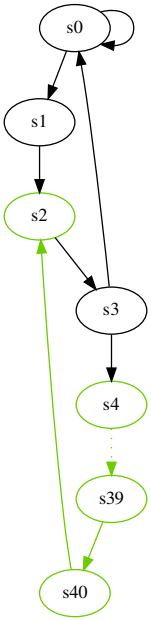
In this example, state S0 is the initial state, which waits for a start signal before transitioning to S1. State S0 and S1 correspond to the initialization code (line 5) and swap code (lines 6-10). While there is a conditional branch at line 6, the code is easily mapped to a multiplexer in hardware, so there is no corresponding branch



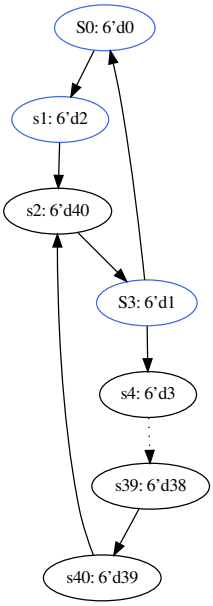

**(a)** Original state diagram




**(b)** Branch-free path 1




**(c)** Branch-free path 2




**(d)** New Encoding

**Figure 3.12:** State assignment example

node in the state graph. States S2 - S40 correspond to the while loop (lines 11-17) [1]. State S3 evaluates the loop condition and returns to S0 if the loop is complete, or transitions to S4 if not. The long branch-free path from S4 - S2 (37 states) corresponds to the loop body (lines 12-16) and is a result of the data dependence between instructions (e.g., lines 14 and 15) and the latency of the instructions. For example, if the divide operation at line 12 is replaced with a multiply operation, the length of the path changes relative to the difference in latency between the two operations.

### 3.6.2 State Assignment

The state transition graph of the example is shown in Figure 3.12a. We first categorize the states, then perform the state encoding on each category separately.

**Categorizing states**: In step 1 of Algorithm 2, we add $S0$ and $S3$, the states with more than one next state, to the set of divergent states. These states are shown in red in Figure 3.12a. In step 2 of Algorithm 2, we find all the branch-free paths that start from successors of $S0$ and $S3$. This step results in finding $path1 =< S1, S2 >$ (Figure 3.12b) and $path2 =< S4, S5, S6, ..., S39, S40, S2 >$ (Figure 3.12c).

In steps 3 and 4, overlapping paths are identified and path refinement (Algorithm 1) is applied. In this example, the two branch-free paths overlap at $S2$, which requires path refinement. After applying the path refinement algorithm, the longer path, $path1$, remains the same while $path2$ will no longer exist since $S1$ becomes the independent state that separate these two paths. $S1$ stores metadata to support the case that the FSM transitions to $path2$ via $S1$ to $S2$.

After the above steps, the only *branch-free* states are the states of $path2$. The remaining states, along with the divergent states, are marked as independent states (blue states in Figure 3.12d). This corresponds to steps 5 and 6 of Algorithm 2.

Now that we have categorized the states, we can perform state assignment on each category according to steps 7 and 8 of Algorithm 2. The result of state assign-

[1] The loop condition, $(0 < A)$, was added so the loop would not be simplified by Vivado. Otherwise, we found Vivado would replace the while loop with the expression "N*result" making the example less interesting for demonstrating our state assignment algorithm.

ment is shown in Figure 3.12d.

### Memory Unit Content

To simplify this example, we assume that the memory unit is sized exactly to fit this FSM, which has a depth of 6 entries and a width of 15-bits. Note that *Next State* and *Path Final State* fields (Figure 3.9) are 6 bits since we require 6 bits to encode all of the states ($floor(\log_2(40states)) = 6$), however, the *Branch Address* field is only 2 bits since there are only 3 states that reside in memory, hence we only require 2 bits to distinguish among these 3 states.

The memory contents for this example is shown in Table 3.2. The first two columns, state label and address, are not part of the memory but have been added to the table to help with understanding which entry corresponds to which address and state. Note that the state encoding shown corresponds to that generated by Algorithm 2 and corresponds to the state values shown in Figure 3.12d. As such, the state encodings may be different from the state label (e.g., S0 = 6'd0, S1 = 6'd2, and S3 = 6'd1). Since the fan-out degree of the states in this example is at most 2, each state will have two corresponding rows in memory, which only requires 1 bit for the *Encoded input* to select the next state. Each memory location containing an *x* indicates that the content of the corresponding field is unused for this state (memory row). This occurs for the state transitions where both current state and next state reside in memory, since the Path Final State and Branch Address entries are only used for the states that belong to branch-free paths. As mentioned earlier, independent states (including divergent states) reside in memory, so any transition between two independent states contains unused memory fields.

### Specialized FSM Block Operation

Next, we describe the operation of our proposed Specialized FSM Block using the example FSM in Figure 3.11. There are four possible operating conditions: transitioning from a memory state to a memory state, from a memory state to a branch-free state, from a branch-free state to a branch-free state on the same path, and from a branch-free state to a memory state. Each of these cases is described below and Figure 3.13 to Figure 3.16 highlight the relevant active portions of the

specialized FSM block.

- **Memory state to memory state transition (e.g., S0 to S1 in Figure 3.12d)**: In this case, illustrated in Figure 3.13 , the FSM block behaves simply like a memory-only FSM implementation. In Figure 3.13, the current state (S0) and encoded input are used to address the memory unit, and the next state (S1) is read from memory. Using the State Control bit from memory, the control unit selects the next state output from the memory to write to the state register. Aside from the State Control bit, the corresponding metadata in the memory is unused.

- **Memory state to branch-free path state (e.g. S3 to S4 in Figure 3.12d)**: In this case, illustrated in Figure 3.14, the control registers, specifically, the Path Final State Register and Branch Address Register, must be updated to control the branch-free path state generation for subsequent cycles. The next-state (i.e., the start of the branch-free path, S4) is loaded into the state registers and the metadata, as described in Section 3.2.4, is loaded into the Path Final State (S2) and Branch Address (S3) registers.

- **Branch-free path state to Branch-free path state on the same path (e.g., S39 to S40 in Figure 3.12d)**: In this case, illustrated in Figure 3.15, the adder in the accumulator unit is used to increment the current state (S39 with encoding 6'd38) to the next state (S40 with encoding 6'd39). The comparator compares the next state with the final state of the path in the Path Final Register (S2 with encoding 6'd40). Since the value of the adder 6'd39 (S40) is not greater than 6'd40 (S2), the accumulator unit and control unit pass the next state (S40) to the state registers.

- **Branch-free path state to Memory state (e.g., S2 to S3 in Figure 3.12d)**: Finally in this case, illustrated in Figure 3.16, the adder unit increments the current state (S2 with encoding 6'd40) and the comparator compares the value of the next state from the adder (6'd41) with the value in the Path Final State Register (S2 with encoding 6'd40). Since the value of the adder is greater than the Path Final State Register, the comparator sets the control signal to select the value in the Branch Address Register (S3 with encoding

6’d1) to send to the state registers. This transitions out of the branch-free path as the next state is used to address the memory unit.

While not a separate case, the transition from S1 to S2 transitions from a memory state to branch-free path state that is not the initial state on the path. This behaves identically to the memory state to branch-free path state transition described above, with the only difference being the initial state that is loaded into the state registers.

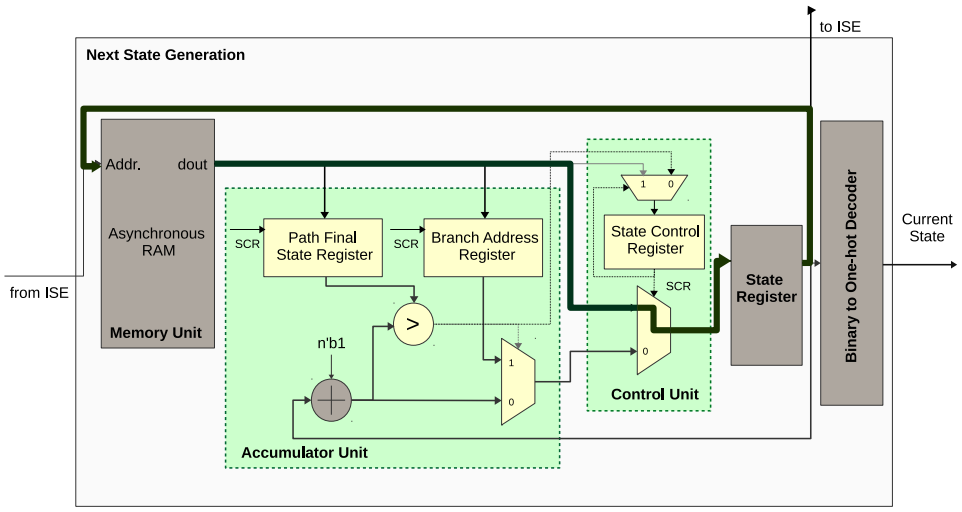



**Figure 3.13:** Next state generation block - transition from a memory state to a memory state

| State | Address [3bits] | Memory Content | | | |
|---|---|---|---|---|---|
| | {CS, Encoded input} | Next State | Final state | Target | Mem/add |
| S0 | {00,0} | 6’d0 | x | x | 1 |
| | {00,1} | 6’d2 | x | x | 1 |
| S1 | {10,0} | 6’d40 | 6’d40 | 2’d1 | 0 |
| | {10,1} | x | x | x | x |
| S3 | {01,0} | 6’d3 | 6’d40 | 2’d1 | 0 |
| | {01,1} | 6’d0 | x | x | 1 |

**Table 3.2:** Memory content

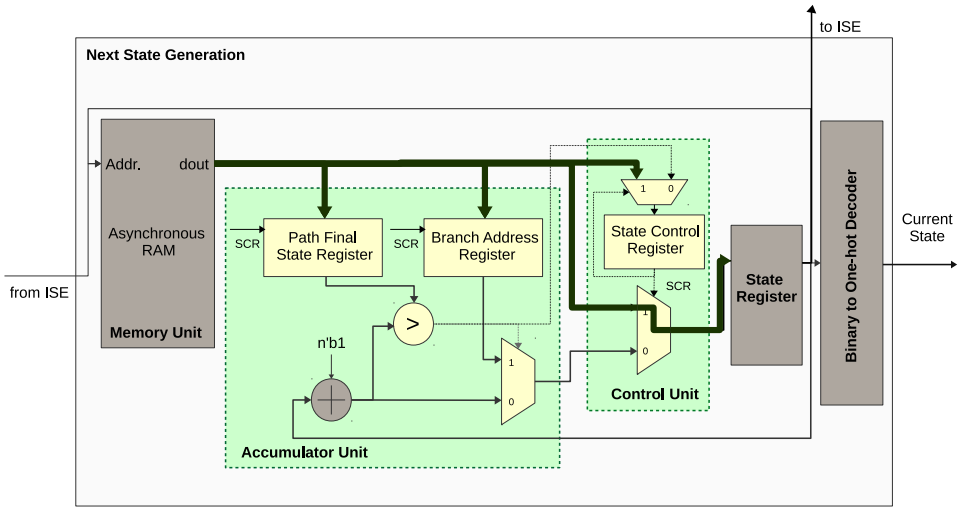



**Figure 3.14:** Next state generation block - transition from a memory state to a branch-free path state

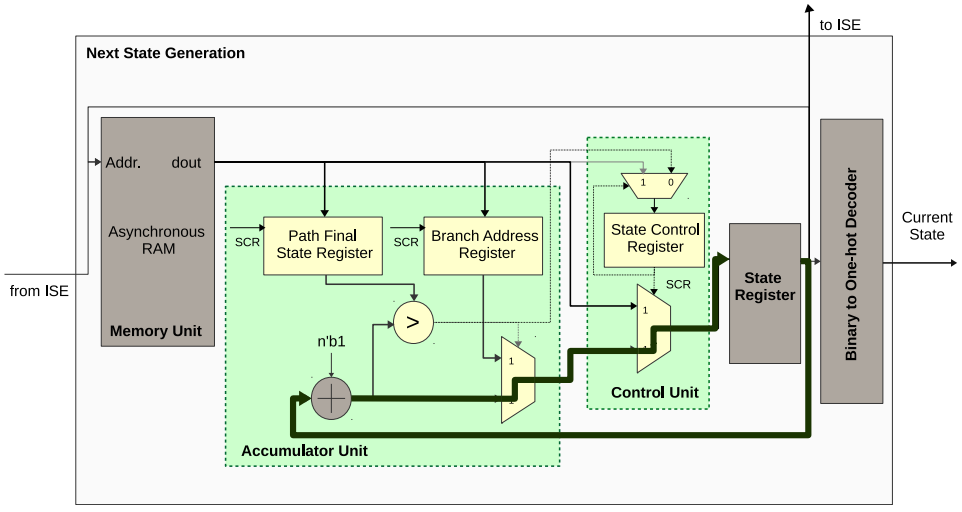



**Figure 3.15:** Next state generation block - transition from a state of a branch-free path to another state on the same path

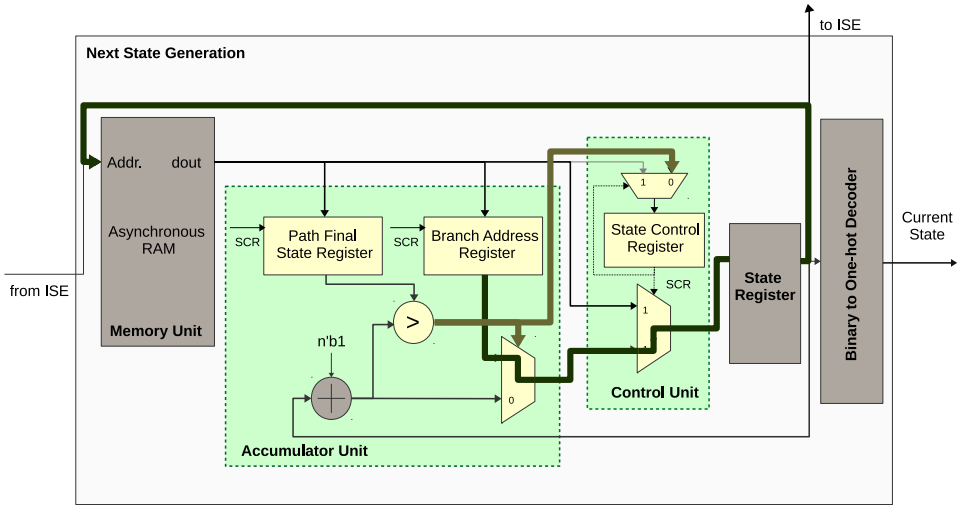



**Figure 3.16:** Next state generation block - transition from the last state of a branch-free path to a memory state

# Chapter 4

# Experimental Methodology

## 4.1 Benchmarks

In this work we use two sets of C/C++ benchmarks to assist with the design and evaluation of our proposed architecture. Both benchmark sets have been developed to be used by HLS tools. The first benchmark set, *MachSuite* [25], is a collection of benchmarks for evaluating accelerator design and customized architectures. The second benchmark sets, *HLS datacenter benchmark*, is developed in our computer architecture group as a joint effort among a group of three students [24]. The HLS datacenter benchmark set consists of high impact functions, in terms of run time, extracted from Lucy [4], SQLite [12], and BZIP [32] benchmarks, and aims to represent benchmarks that may be commonly run in a datacenter. Some parts of these benchmarks have been re-written to replace C/C++ features that are not supported by Vivado HLS.

Table 4.1 shows the number of lines of C/C++ code for the benchmarks in each benchmark set. This is used to highlight the size and complexity of the benchmarks to better understand the resulting FSMs from HLS.

We convert the benchmarks from C/C++ to Verilog HDL using Vivado HLS. For the main part of our evaluation we use the default HLS directives provided in Machsuite to improve the quality of the generated Verilog code, however, the default HLS directives might not necessarily lead to the most optimized design. In order to directly evaluate the impact of HLS optimization for certain optimization

| Benchmark | LOC |
|---|---|
| backprop | 159 |
| aes | 167 |
| viterbi | 44 |
| spmv_crs | 34 |
| spmv_ellpack | 34 |
| nw | 98 |
| bfs_bulk | 64 |
| bfs_queue | 71 |
| fft_transpose | 363 |
| fftstrided | 43 |
| sort_merge | 57 |
| sort_radix | 116 |
| kmp | 52 |
| stencil3d | 46 |
| stencil2d | 39 |
| md_knn | 71 |
| md_grid | 78 |
| gemm_ncubed | 41 |
| gemm_blocked | 43 |

**(a)** Machsuite Benchmarks

| Benchmark | LOC |
|---|---|
| bzip | 747 |
| lucy_sn | 78 |
| lucy_sv | 82 |
| lucy_sa | 66 |
| sqlite_ln | 561 |
| sqlite_gt | 410 |

**(b)** HLS Datacenter Benchmarks

**Table 4.1:** Number of lines of actual C code, excluding comments, for the evaluated benchmarks

goals (such as area, delay, and area-delay product) on the generated FSMs, we use the model described in Lo et al. [18]. This work uses sequential model-based optimization methods to automatically select the set of HLS directives that optimize the design for different optimization goals. We use the data provided by Lo et al. to obtain the HLS directive settings that minimize the area-delay product of the generated RTL design for the *aes*, *backprop* and *sort radix* benchmarks. The result of this analysis is provided in Section 5.6

| Benchmark | States | Inputs | Max fanout |
|---|---|---|---|
| aes_fsm1 | 47 | 6 | 2 |
| aes_fsm2 | 76 | 14 | 2 |
| bckp_fsm1 | 11 | 11 | 2 |
| bckp_fsm2 | 158 | 10 | 2 |
| bckp_fsm3 | 69 | 6 | 2 |
| bfs_b_fsm | 8 | 7 | 3 |
| bfs_q_fsm | 8 | 6 | 2 |
| fft_st_fsm | 24 | 5 | 2 |
| fft_tr_fsm1 | 17 | 8 | 2 |
| fft_tr_fsm2 | 24 | 6 | 2 |
| fft_tr_fsm3 | 219 | 14 | 2 |
| fft_tr_fsm4 | 10 | 6 | 2 |
| fft_tr_fsm5 | 66 | 5 | 2 |
| gemm_fsm1 | 10 | 8 | 2 |
| kmp_fsm1 | 7 | 4 | 2 |
| kmp_fsm2 | 10 | 6 | 2 |
| md_gr_fsm | 15 | 10 | 2 |
| md_knn_fsm | 98 | 5 | 2 |
| sort_m_fsm1 | 4 | 5 | 2 |
| sort_m_fsm2 | 7 | 5 | 2 |
| sort_r_fsm1 | 15 | 11 | 2 |
| sort_r_fsm2 | 6 | 4 | 2 |
| sort_r_fsm3 | 6 | 4 | 2 |
| spmv_crs_fsm | 10 | 6 | 2 |
| smpv_elpk_fsm | 9 | 6 | 2 |
| stencil_fsm | 4 | 4 | 2 |
| viterbi_fsm | 8 | 6 | 2 |

**Table 4.2:** Characteristics of the FSMs extracted from MachSuite

## 4.2 FSM Extraction

To evaluate our proposed mix-grained architecture, we must extract the finite state machines from each benchmark. This is achieved as follows: We use Yosys synthesis tool [37] to synthesize each benchmark to an RTL netlist. We then use the *FSM detection* and *FSM extraction* passes provided in Yosys to detect and extract

| Benchmark | States | Inputs | Max fanout |
|---|---|---|---|
| lucy_sh_fsm | 71 | 3 | 2 |
| sql_ln_fsm1 | 508 | 56 | 4 |
| sql_ln_fsm2 | 7 | 6 | 3 |
| sql_ln_fsm3 | 5 | 6 | 3 |
| sql_ln_fsm4 | 10 | 10 | 3 |
| sql_ln_fsm5 | 4 | 4 | 2 |
| sql_ln_fsm6 | 4 | 4 | 2 |
| lucy_sn_fsm | 25 | 5 | 2 |
| lucy_sv_fsm | 12 | 10 | 4 |
| bzip_fsm1 | 72 | 19 | 3 |
| bzip_fsm2 | 41 | 11 | 2 |
| bzip_fsm3 | 67 | 28 | 4 |
| bzip_fsm4 | 17 | 9 | 3 |
| bzip_fsm5 | 43 | 4 | 2 |
| bzip_fsm6 | 61 | 19 | 3 |
| bzip_fsm7 | 36 | 13 | 2 |
| bzip_fsm8 | 117 | 34 | 3 |
| sql_gt_fsm1 | 61 | 48 | 4 |
| sql_gt_fsm2 | 12 | 9 | 2 |

**Table 4.3:** Characteristics of the FSMs extracted from HLS datacenter benchmarks

the state machines from the rest of the design. These passes implement an algorithm similar to the algorithm proposed in [29] to extract the FSM from a flatten netlist. The extracted FSM is in KISS [27] format, a simple format to store the FSM transition table. We have developed an FSM generator in C++ which, given a finite-state machine described in KISS format, generates the Verilog HDL code that describes this FSM. For the purpose of this work, we were only interested in the RTL code of the next state calculation logic, hence our FSM generator only generates the RTL design for the next state calculation logic and does not include the output calculation logic in the generated design. Using this flow we are able to extract the FSM from any given benchmark and generate a stand-alone RTL design that describes this state machine.

The statistics of the FSM that we have extracted from the MachSuite and data

center benchmarks are shown in Table 4.2 and Table 4.3.

## 4.3 Area and Delay Model

### Next State Generation Block Area Model

To model the next state generation block, which correspond to the coarse-grained part of the FSM architecture of Figure 3.5, we have described the architecture of this block in Verilog HDL. This excludes the area model used for Input Sequence Encoder which is described in next section. The memory unit is modelled using the Artisan synchronous SRAM compiler [5]. As described in Section 3.2.4, the Memory Unit in Figure 3.8 is an asynchronous memory. However, since we did not have access to an asynchronous SRAM memory compiler, we used a synchronous memory unit to model the area. We believe that the area of the asynchronous memory will be comparable to the synchronous memory unit. However, a small error in the area estimation will have a minimal affect on the total area of the proposed specialized FSM architecture, since the Next state generation block counts for less than half of the block area for small FSMs, and is much less than half for the larger FSMs.

The RTL design has been synthesized using the Synopsis Design Compiler vH-2013.03-SP5-2 [33] with the TSMC 65nm library. The area estimations presented in this dissertation are pre place-and-route. We estimate the routing area of the next state generation block, which is not calculated by the Synopsys design compiler as follows: We exclude the area of the RAM (since the internal routing has already been modelled by the SRAM compiler), then we multiply the area of the remaining units, which is reported by design compiler, by a factor of 2X. Note that by using this approach, we are overestimating the area of the block, since the routing inside the next state generation unit is very limited. Thus, the presented area estimations are conservative.

### Input Sequence Encoder Area Model

We have developed an input sequence encoder generator in C++. It takes the FSM described in KISS format and generates the Verilog HDL that implements this

encoder, as described in Section 3.5.2.

The RTL design for the input sequence encoder is then implemented onto the FPGA soft logic. We use the FPGA architecture *k6_frac_N10_40nm* provided in VTR [19] to model the area of the input sequence encoder, and map the input sequence encoder described in verilog to the FPGA soft logic. We then use the following formula, which is also used by VTR, to convert the logic and routing area reported by VTR in Minimum Width Transistor Area (MWTA) to $um^2$:

$$1 * MWTA = 70 * (\lambda)^2$$

Where $\lambda$ is equal to 65nm.

**Specialized FSM Architecture Delay Model**

Next we describe the delay model used for the proposed specialized FSM architecture which consists the delay of both input sequence encoder and next state generation block. Looking at Figure 3.8, the critical path delay reported by design compiler for the next state generation block starts from the output of the state register through the adder and two multiplexers back to the input of the state register. Note that, for the scenario when the next state calculation is solely calculated using the accumulator unit, the total critical path delay of the FSM architecture is equal to the critical path delay of the next state generation block. However, for the case where the next state calculation is performed through input sequence encoder and memory unit, the output of the state register is fed back to the input of the input sequence encoder. Therefore the critical path delay of the input sequence encoder along with the critical path delay of the next state generation block form the total delay of the architecture.

The delay of the input sequence encoder is obtained from VTR by mapping the encoder onto the baseline architecture. The delay values for the next state generation block are obtained from the design compiler. To account for the effect of the place and route on the delay values, we use the same experience-based estimation approach stated in [45], which suggests on average paths degrade by a factor of 1.6X after layout.

Note that we provide an optional binary-to-onehot decoder at the output of the

FSM block (Figure 3.8). This decoder is located after the state registers, hence after obtaining the total critical path of design as mentioned above, we also add the latency of this decoder to the total critical path of the specialized FSM architecture.

#### Baseline FPGA architecture

The baseline FPGA architecture is also *k6_frac_N10_40nm*. We selected the simple architecture without any hard block as the baseline to minimize the area overhead of unused hard blocks that the FSM will not benefit from.

## 4.4 CAD Flow

To synthesize our benchmarks onto the FPGA baseline and our proposed architecture, we use VTR 7.0 [19]. VTR provides the full synthesis, technology mapping, placement, and routing steps required to compile the proposed next state generation hard block and input sequence encoder soft block onto the baseline FPGA architecture.

## 4.5 Mapping to the Next State Generation Block

As described in Section 3.5.3, for a given state machine to fit into the next state generation block, there are two required conditions that must be met: (1) the number of bits required for the state encoding should not exceed the maximum bit-width of the adder, 2) the number of states that reside in memory must be less than the memory size.

To evaluate these two conditions, we first apply the state assignment algorithm, Algorithm 2, on the given FSM. After performing the state encoding, we will have the number of state bits required to encode the state values and the total number of states that will be assigned to the memory unit. In case any of these two requirements are not met, we can use the FSM partitioning technique described in Section 3.3 to map the FSM to two or more combined fracturable blocks.

# Chapter 5

# Experimental Results

This chapter presents and discusses our experimental results. We first use the result of applying our state assignment technique on the finite state machines extracted from MachSuite and the datacenter benchmarks to explain the sizing of the FSM block. We then evaluate the overall area and delay improvement of our proposed architecture over these benchmarks. We also provide the detail characteristics of each FSM to fully explain the variation in the result of area/delay improvement over these benchmarks. We then demonstrate the outcome of applying HLS optimization to three MachSuite benchmarks on the characteristics of the generated state machines. We finally assess the functionality of the FM partitioning algorithm on an FSM that does not fit into one FSM block and measure the overhead of our proposed modifications to support fracturable FSM blocks.

## 5.1 Next State Generation Block Size

In this section we explain the reason behind the size decisions for elements of the next state generation block. The best sizing for the FSM block will accommodate the common FSM size, while reducing the amount of wasted resources if the common FSMs are smaller than the selected FSM block area.

Figure 5.1 shows the area breakdown of each unit of the next state generation block as a fraction of the total area for the block configuration given in Table 5.1. As can be seen in this figure, the memory block is the main contributor to the next

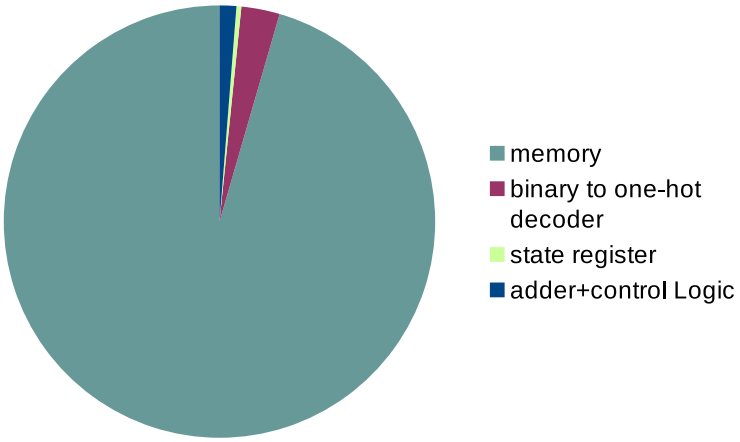



**Figure 5.1:** Area breakdown of the coarse-grained fabric

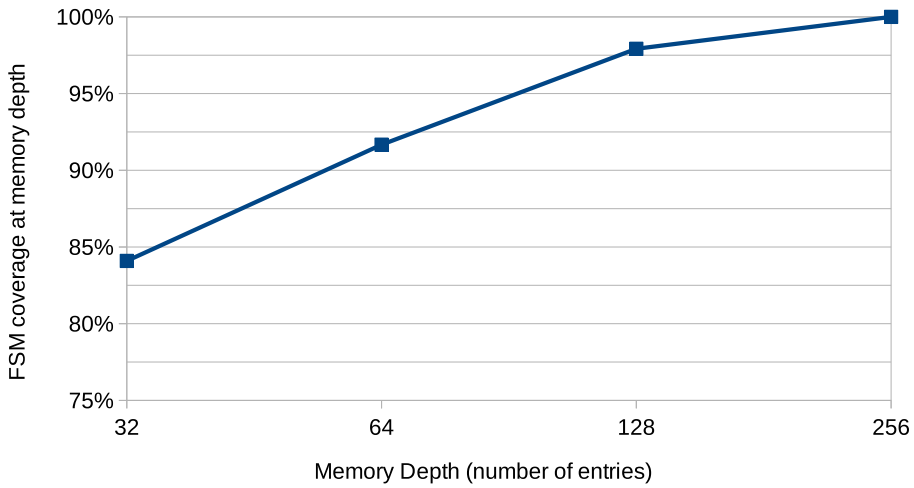



**Figure 5.2:** FSM coverage vs. memory depth in number of entries. Approximately 98% of the evaluated FSMs fit into a memory with a depth of 128 entries.

state generation block area. We have measured the area breakdown of the block for various block configurations by sweeping the memory size, however, the memory unit always remains the main contributor to the block area since the area of the remaining units also scale accordingly as the memory size varies. Therefore, it is

important to select a memory unit with the proper size to minimize the total area of our proposed FSM block.

We have collected the required memory depth, in terms of number of entries (independent states), for our evaluated benchmarks. Figure 5.2 presents the fraction of the FSMs that will fit in a certain memory depth of 32, 64, 128, and 256 entries. For our workloads, 98% of the FSMs fit into a depth of 128. Thus, for the remainder of our evaluation, we selected a memory size with a depth of 128 entries to accommodate the common FSM sizes.

The second design decision is the bit-width of the adder, control registers, and encoding bits. To choose these values, we need to answer the following question: What are the total number of states for a state machine that uses all the 128 memory entries? To answer this question we require two data points: (1) what percentage of the states are typically allocated in memory, and (2) what is the maximum fan-out degree over our evaluated FSM. The second question helps determine how many memory rows are needed for each state.

The answer to the first question is shown in Figure 3.6. On average approximately 18% of the states reside in memory. The second question can be answered by looking at Figure 3.7 which shows the maximum number of fan-out per state is equal to 4. Therefore, given a memory unit that has 4 memory rows associated with each state, and where the number of memory states is 20% of the total number of states, the total number of states in an FSM that can map to this memory unit is equal to ($\frac{\text{128 states}}{\text{4 rows per state}} \times \frac{1}{\text{20\% of total states}} = 160$ states). Hence we use 8 bits to represent the states in such a state machine. For any state machines that require more bits for the state encoding, there is a high chance that the memory size will not be able to accommodate all the states.

Using the format of memory content presented in Figure 3.9, the memory width should be equal to (size of Next State value + size of Path Final State value + size of State Control value) which is $8+8+5+1 = 22$ bits. Putting it all together, the size of the units in the next state generation block can be seen in Table 5.1

| Total Memory size | 128x22 bits |
|---|---|
| Adder size | 8 bits |
| State Register size | 8 bits |
| Encoded Input Sequence size | 2 bits |

**Table 5.1:** The sizing configuration of the elements of the next state generation block

## 5.2 Area Improvement

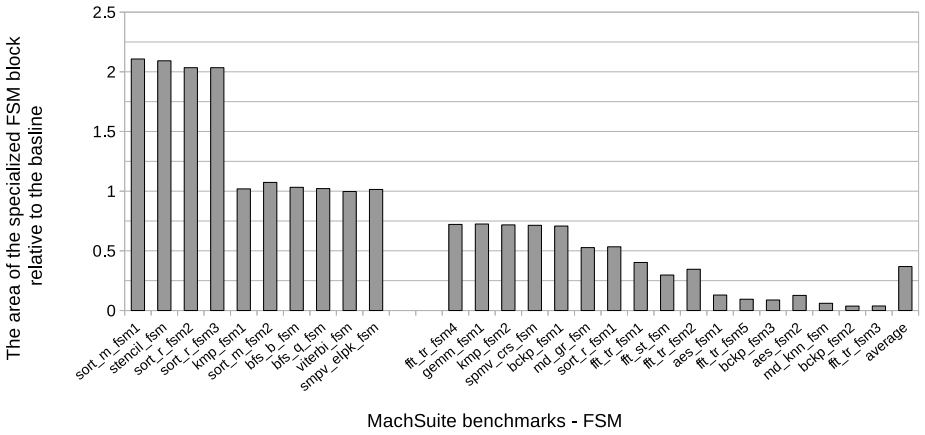
This section presents the improvement in FSM area using our proposed specialized FSM block compared to the baseline LUT-based FSM implementation. The area of the next state generation block for the configuration given in Section 5.1 is equal to 15662 $um^2$ which is calculated as described in Chapter 4. The area improvement for the MachSuite and HLS datacenter benchmarks is presented in Figure 5.3. The subsequent figure, Figure 5.4, presents a breakdown of state categories to assist with analyzing the variation in area savings. The breakdown of state categories is collected after performing state assignment on each FSM.

In Figure 5.3, the x-axis shows the FSMs extracted from the benchmark sets and the y-axis shows the relative area of our proposed architecture compared to the baseline LUT-based implementation. The gap between the bars on the x-axis separates the FSMs that have less than 10 states (on the left) from the FSMs with more than 10 states (on the right). In the extreme cases where the FSM only has a few states, less than 10, the number of states on the branch-free paths and the number of states to be stored in memory are so limited that it does not justify the area overhead of using the FSM hard block with a memory depth of 128. This issue can be addressed with two different approaches. First, a simple predictor based on the FSM size could be used during the synthesis to decide whether the FSM should be mapped to the proposed FSM hard block or should be implemented using the soft logic on FPGAs. Second, the FSM block can be sized down to accommodate smaller FSMs. However, this also results in a lower percentage of FSMs fitting into a single block. The FSMs that do not fit into a single block will be split over multiple fracturable blocks. The overheads of having fracturable blocks are discussed in Section 5.7.

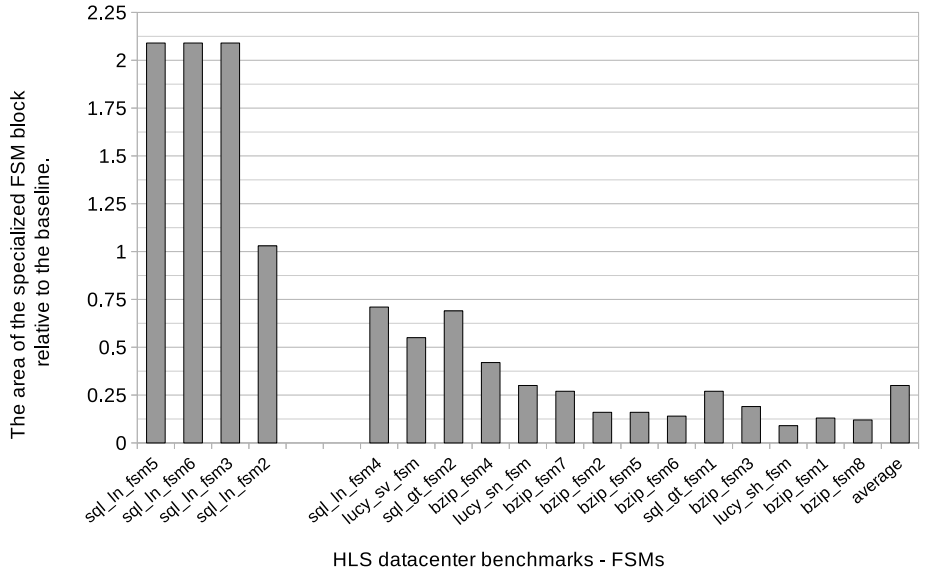
As shown in Figure 5.3, in average the area of our proposed architecture is approximately 36% of the baseline FPGA architecture for the MachSuite benchmarks, and is approximately 30% of the baseline area for the HLS datacenter benchmarks. These averages are not including the benchmarks that have FSMs with fewer than 10 states.

The main trend that can be seen is that the area improvement increases as the FSM size increases. This is due to the increase in the amount of soft logic that is required to implement the baseline FSM, which is replaced by our fixed-size specialized FSM block.

Figure 5.4 can be used to help explain the area improvements for different FSMs. The x-axis shows the FSMs from our evaluated benchmarks and the y-axis shows the total number of states as a breakdown of branch-free and memory states. The main trend that we see is that as the number of states that can be mapped to the branch-free paths increases, the area savings also increase. For the state machines that have the same number of states but a different area improvement, the complexity of the input sequence encoder is the main reason for the area difference. As the number of states that need to be stored in memory increases, the logic to implement the input sequence encoder will be more complex, resulting in having a larger area. This can be seen for *bzip_fsm6* and *sql_gt_fsm1*. These benchmarks have the same number of states (61 states), however, the total number of states that reside in memory for *sql_gt_fsm1* is equal to 30 while it is only 10 for *bzip_fsm6*. Consequently, as shown in Figure 5.3, *bzip_fsm_6* has a smaller area (14% of the baseline) compared to *sql_gt_fsm1* (27% of the baseline). However, one exception is with benchmarks *lucy_sv_fsm* and *sql_gt_fsm2* where benchmark *lucy_sv_fsm* has more memory states and better area improvement than *sql_gt_fsm2*. We expect that this is due to the higher complexity of the next state calculation logic for benchmark *lucy_sv_fsm* than benchmark *sql_gt_fsm2*, which results in a greater area reduction when mapping to a simple memory lookup. Further analysis to this scenario is left to future work.




**(a)** MachSuite

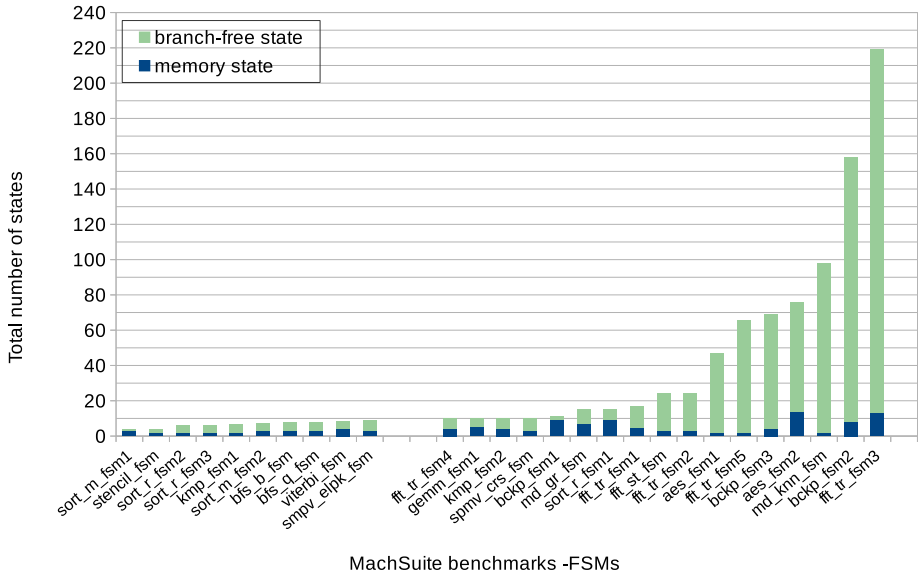



**(b)** HLS datacenter

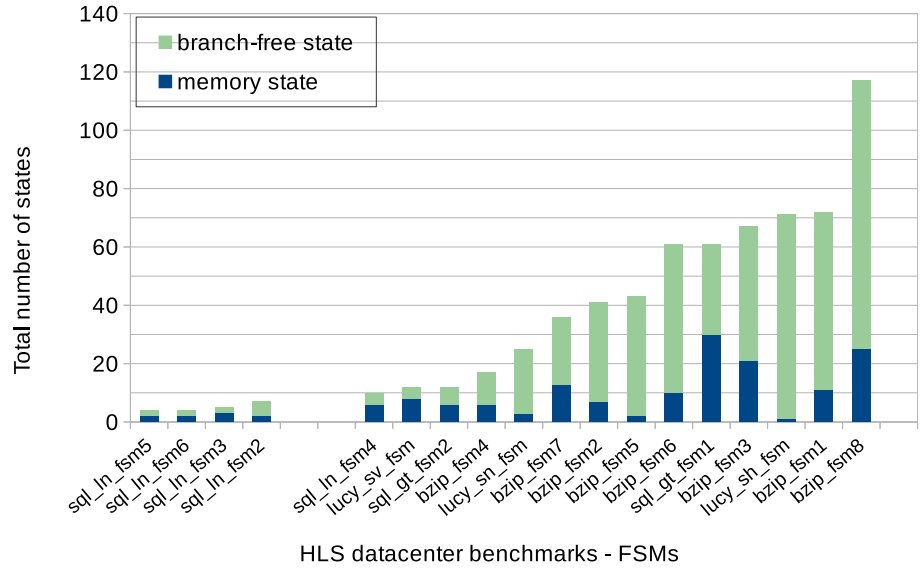
**Figure 5.3:** Area improvement of the specialized FSM architecture, which includes the area of the input sequence encoder and next state generation block relative to the baseline

## 5.3 Delay Improvement

The input to output delay of the next state generation block for the configuration given in Section 5.1 is equal to 0.5 ns, which is calculated as described in Chapter 4. The delay improvement achieved by our specialized FSM block is shown in




**(a)** MachSuite




**(b)** HLS datacenter

**Figure 5.4:** FSM size along with the breakdown of the states that are part of branch-free paths and states that reside in memory

Figure 5.5 for the evaluated benchmarks. The x-axis shows the FSMs from the different benchmarks and the y-axis shows the critical path, relative to the baseline.

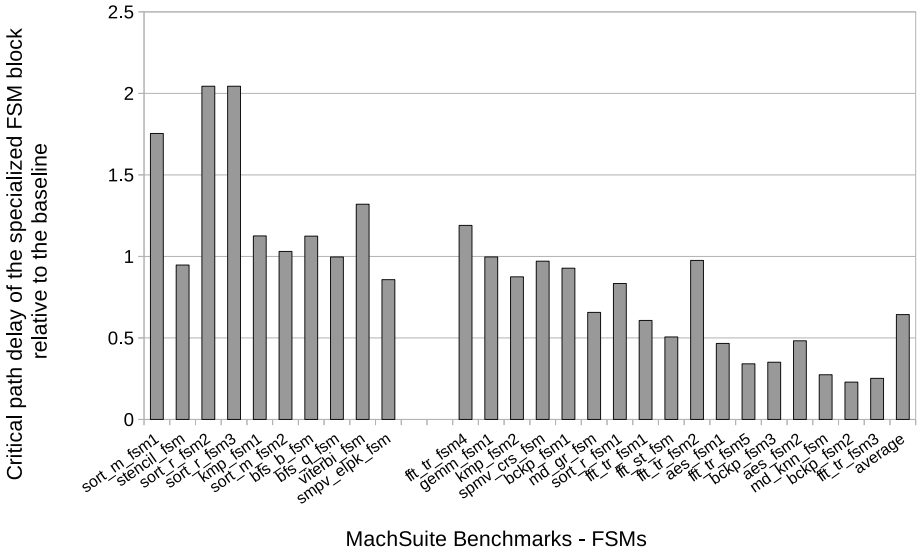
As above, the FSMs with less than 10 states are separated from the FSMs with more than 10 states by a gap. As with area savings, the FSMs with at least 10 states will benefit from our specialized hard block, and the critical path delay improves as the size and complexity of the FSM increases. This is due to the fact that, for smaller FSMs, the overhead of the extra control logic in the FSM blocks is not negligible compared to the critical path delay of the LUT-based portion of the FSM.

Similar to the area results, the complexity of the input sequence encoder is a large contributor to the critical path of the total design, which is indicated by the number of states that are mapped to memory, as shown in Figure 5.4.

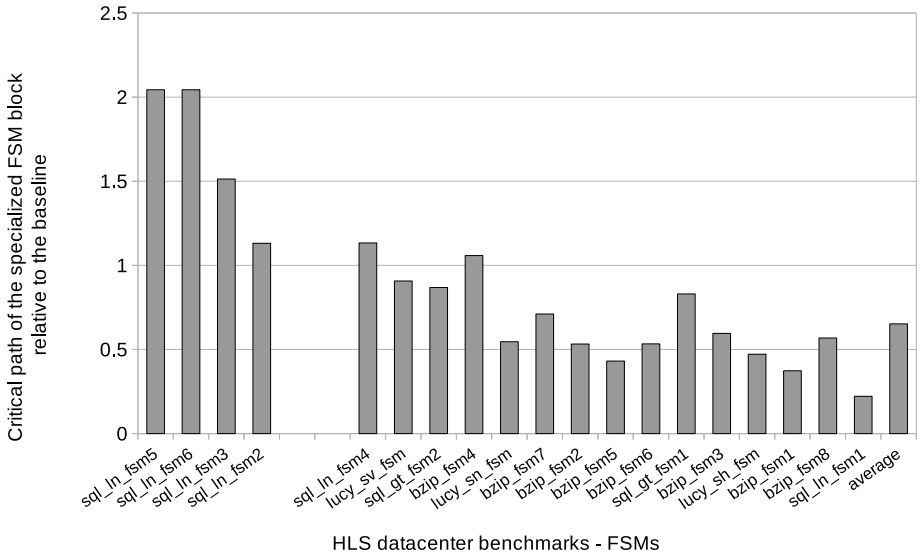
## 5.4 Resource Usage of the Mix-Grained Architecture

Figure 5.6 illustrates the area of each unit as a fraction of the total area of the mix-grained architecture for the same workloads presented in Figure 5.3. As this result shows, about 50% of the area is consumed by the input sequence encoder. This amount varies among the benchmarks as size of the FSM and more specifically, number of states that reside in memory varies. However, in addition to the number of memory states, the complexity of the boolean function that defines the transition between states also affects the complexity and size of the input sequence encoder. As can be seen in Figure 5.4a, number of memory states among MachSuite benchmarks is mainly less than 10, independent of the FSM size. This results in small variation in size of the input sequence encoder among MachSuite benchmarks. However, for the Datacenter benchmarks (Figure 5.4b), there is a higher variation in number of memory states among different benchmarks, hence there is more variation in size of the input sequence encoder for these benchmarks as well.

The area of the hard block, consisting of the memory, adder unit, and output decoder is always fixed. This explain the increase in area savings for the larger FSMs, since the overhead of the control logic in the hard block will be negligible compared to the input encoder implemented using the FPGA soft logic.

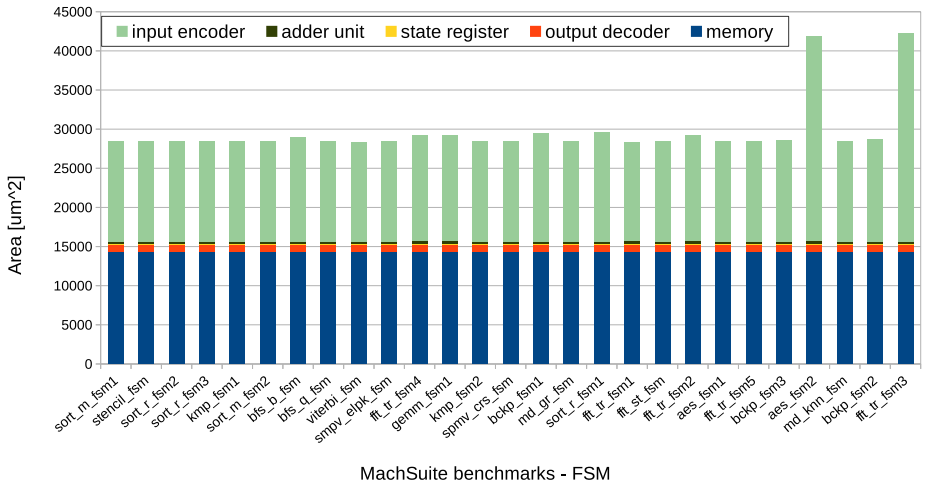



**(a)** MachSuite

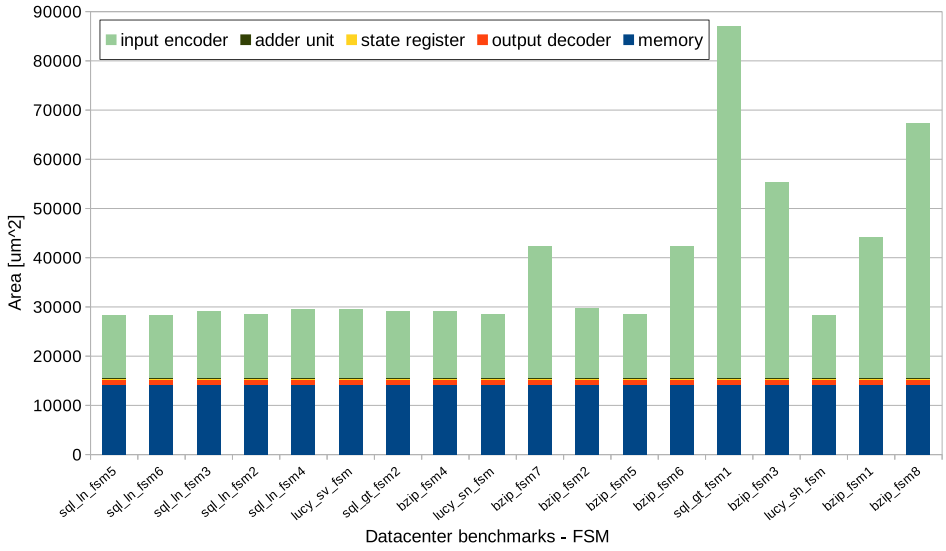



**(b)** HLS datacenter

**Figure 5.5:** Critical path delay improvement of the specialized FSM architecture which includes the critical path of the input sequence encoder and next state generation block relative to the baseline




**(a)** MachSuite




**(b)** HLS datacenter

**Figure 5.6:** Area breakdown of the mix-grained FSM architecture for the FSMs extracted from the evaluated benchmarks

## 5.5 FSM Area

Along with the area improvement of the FSMs, we are also interested in the fraction of the FSM next state calculation area to the total application design area (control plus data-path). However, for most of the evaluated benchmarks, the generation of IP (Intellectual Property) cores as part of the design by Vivado HLS limits the

| benchmark | area percentage of the FSM next state calculation logic | total number of the states |
|---|---|---|
| sqlite_ln | 11.27% | 508 |
| sqlite_gt | 9.12% | 73 |

**Table 5.2:** Fraction of next state calculation logic area to total design area for *sqlite_lookupName* and *sqlite_getToken* functions from the HLS datacenter benchmark set

ability to implement the whole design onto the baseline FPGA using VTR. Therefore, we were unable to measure the total area of the designs for these benchmarks, since Vivado synthesis tool only provides the resource utilization of the final synthesized design, not the total area. We were able to measure this fraction for the *SQLite* benchmark (from the datacenter benchmark set) which does not contain any IP cores. The percentage of the area for the next state calculation logic for two functions of *SQLite* benchmark is shown in Table 5.2. On average for these two functions, the next state calculation logic area is approximately 10.19% of the total design area. We leave evaluating the total area for all benchmarks, including IP cores, to future work.

## 5.6 Impact of HLS Directives on the Generated FSMs

To evaluate the impact of HLS optimization on FSM characteristics, we have applied a set of HLS directives that minimize the area-delay product of the *aes*, *backprop*, and *sort radix* benchmarks. In Chapter 4, we have described how these HLS settings have been obtained.

The impact of applying HLS directives on the three mentioned MachSuite benchmarks is shown in Figure 5.7, and is averaged across these benchmarks. These benchmarks were arbitrarily chosen to show the impact of HLS directives. The x-axis is labelled by the number of fan-outs per state, and the y-axis indicates the fraction of total states that have the corresponding fan-out degree. As can be seen, on average, the optimized designs (opt) have a higher number of branch-free paths than the non-optimized designs (no-opt), e.g., fan-out 1 is higher for the

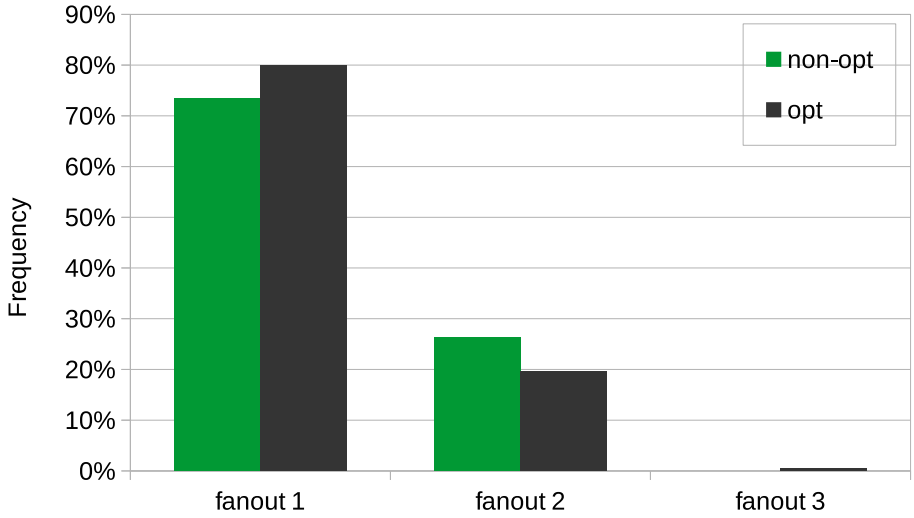
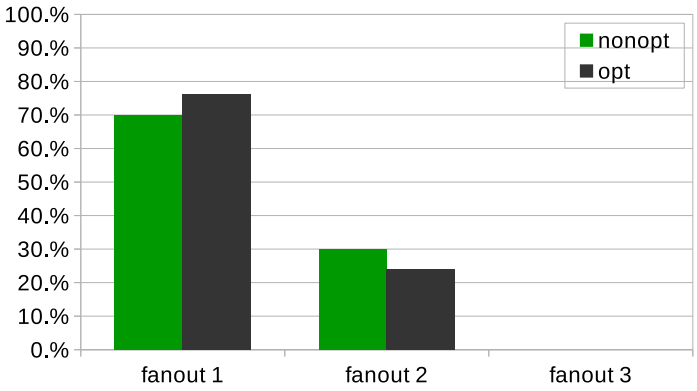


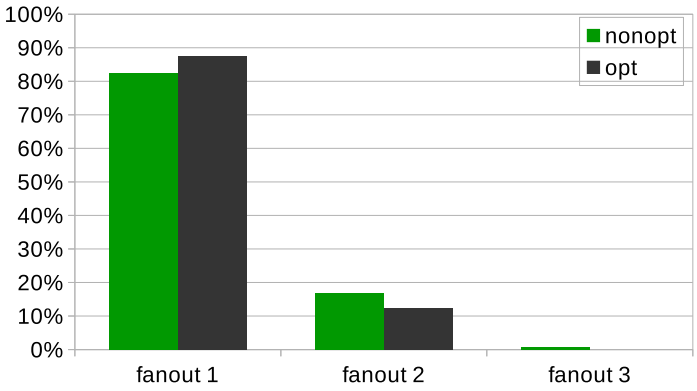

**Figure 5.7:** Impact of applying HLS optimization directives on *backprop*, *aes*, and *radix sort* benchmarks from MachSuite. The x-axis shows the number of reachable next states per state. For example, fan-out 1 is an indicator for states that only have one next state. This figure shows that the optimization increases the percentage of branch-free path.

pragma optimized (opt) versus non-optimized (non-opt) versions. The affect of applying the HLS directive on each individual benchmarks as an average over all the FSMs extracted from each benchmark is shown in Figure 5.8. The x-axis and y-axis are the same as described for Figure 5.7. By looking at the impact of HLS directives on each individual benchmark, we observed that the HLS directives do not necessarily change the ratio of fan-out degrees 1 and 2, however, they result in generating more FSMs for different parts of the same design which, in the case of these three benchmarks, contain a higher ratio of branch-free paths.

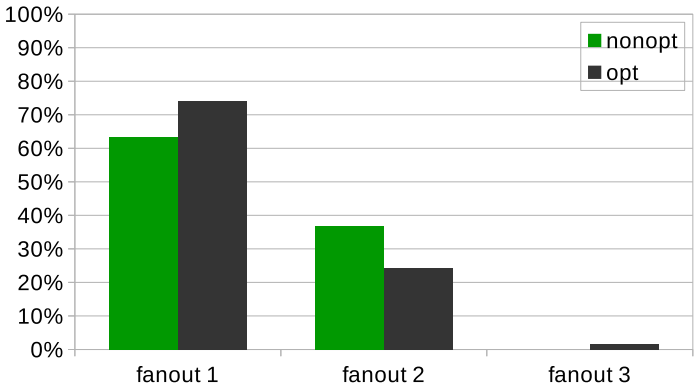
Many of the HLS directives attempt to exploit more parallelism, for example, by loop unrolling and loop pipelining. In these cases, it results in an increase in the number of states to generate the control signals for the unrolled and pipeline loops, adding more branch-free states in between divergent states used to control the loops.




**(a)** backprop




**(b)** aes




**(c)** sort radix

**Figure 5.8:** Impact of applying HLS optimization directives on *aes*, *backprop*, and *radix sort* benchmarks from MachSuite

| Benchmark | States | Required state bits | Memory states | Required mem. depth | Max fan-out |
|---|---|---|---|---|---|
| sql_ln_fsm1 | 508 | 9 | 40 | 160 | 4 |

**Table 5.3:** Block size checking for the *sql_ln_fsm1* indicating that it does not fit to one FSM block, since it requires larger memory unit and more state bits than what is provided by the FSM block.

## 5.7 Efficiency of the Fracturable FSM Block

In this section we evaluate the efficiency of our proposed solution for the scenario where a large FSM does not map to one FSM block. We perform analysis on an FSM with 508 states, which is extracted from the *sqlite_lookupName* function. The corresponding FSM is named *sql_ln_fsm1*. Below we describe the the required steps for mapping this FSM to two fracturable FSM blocks.

**Block size**: Table 5.3 shows the block size information for the *sql_ln_fsm1* FSM. The FSM has 508 states, which requires 9 state bits, thus it is too large to map to our FSM block with only 8 state bits and an 8 bit-wide adder. Additionally, the result of the state encoding shows that 40 states must be mapped to the memory unit. The maximum number of fan-out per state for this FSM is equal to 4, thus we need 2 bits to represent the *encoded input*, which allows each memory states to have a maximum of four corresponding memory rows to store the data for potential state transitions. Therefore, a memory unit with 160 entries is required to accommodate this FSM, which will not fit into our proposed FSM block with 128 entries.

**Partitioning**: Table 5.4 describes the result of applying the Fiduccia-Matheyses partitioning algorithm on sql_ln_fsm1, as described in Section 3.3.1, and then re-performing the state assignment on each of these partitions. The first row indicates the number of states that are required to be mapped to the memory unit in each FSM partition. The second row presents the overhead of partitioning in terms of the number of states that are required to be mapped to the memory to store the information corresponding to the transitions across the the fracturable blocks. The third and fourth row show the total number of memory states for each FSM partition and the required memory size to accommodate them.

The values for the refined required memory sizes indicate that the partitioning

can result in an unbalanced division of the required memory size between partitions. The partitioning algorithm aims to minimize the cut set value between two partitions, however, it is possible to have different number of branches within each partition. A more sophisticated partitioning algorithm can aim to also balance this number in addition to minimizing the cut set value to better utilize each fracturable block. Evaluating the impact of different partitioning algorithms is left to future work.

**Area saving**:Figure 5.9 shows the area overhead of using a fracturable FSM block to map a large FSM as opposed to having one large specialized FSM block to fit the FSM. LUT-based implementation of the FSM in FPGA soft logic is used as the baseline. The area overhead due to making the FSM block fracturable is negligible compared to the area improvement gained by mapping the FSM to the FSM hard block.

The results of splitting a large FSM over two fracturable blocks, Figure 5.9, show the efficiency of this approach for a medium size FSM block (a memory unit with 256 entries). As shown in Table 5.4, partitioning an FSM results in storing additional states in memory. For the smaller FSM blocks, e.g. a memory unit with 128 entries, there are only 32 states that can be stored in memory (assuming each state has 4 memory locations for 4 potential next states). This memory size offers a very limited space for storing states. By adding the overhead of additional states that are caused by partitioning, this memory unit can easily become full which leads to requiring more than two fracturable blocks to accommodate a given medium size FSM. This might result in extra overhead that is more than the amount shown in Figure 5.9. In this work we only look at splitting the FSM across two fsm blocks. Evaluating the efficiency of mapping an FSM over multiple blocks is left to future work.

| | Partition A | Partition B |
|---|---|---|
| Initial number of memory states | 24 | 16 |
| Number of overhead memory states | 6 | 6 |
| Refined number of memory states | 30 | 22 |
| Refined required memory size | 120 | 88 |

**Table 5.4:** Memory unit size requirement for each partition after partitioning the FSM. Row 4 indicates that FSM partition A requires a memory unit of size 120 and FSM partition B requires a memory unit of size 88, thus they both can be mapped to a fracturable FSM block.

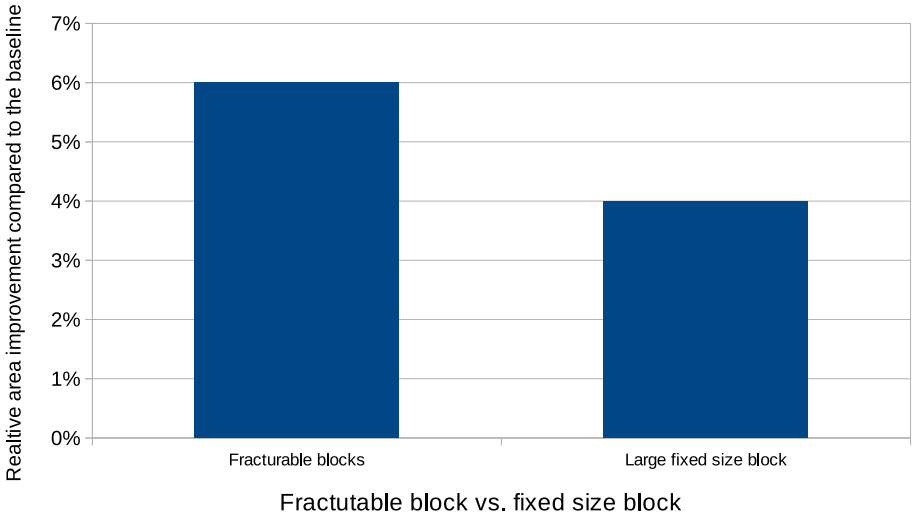



**Figure 5.9:** Area overhead of using a fracturable FSM block to map a large FSM as opposed to having one large specialized FSM block to fit the FSM. The overhead due to making the FSM block fracturable is negligible compared to the area improvement gained by mapping the FSM to the hard FSM block.

# Chapter 6

# Related Work

This chapter summarizes and contrasts the work done in this dissertation against related work on FPGA hard blocks and configurable FSM architectures.

There is a large body of work looking at using specialized hard blocks as part of the FPGA's architecture. Wilton et al. [36] examines the architecture of FPGAs containing coarse-grained memory blocks, Langhammer et al. [16] proposes DSP blocks that support floating point operation, and all modern FPGAs now contain specialized hard blocks as part of their architecture [[43], [2], [41], [40], [1]]. Similar to our work, these works share a common goal of introducing specialized blocks to the FPGA's architecture that perform a set of specific tasks in a more efficient manner in terms of area, performance, and power. However, they mainly look at improving the complex operations and functional units that are common in the data-path part of hardware designs, whereas we propose a hard block that is designed to better implement the control-path portion of digital systems.

Garcia-Vargas et al. [10] proposes to use block RAMs provided in modern FPGA architecture to implement FSMs. This work looks at implementing the next state/output calculation for every state using memory. They reduce the size of the memory by multiplexing the FSM inputs to choose the set of active inputs at each state. Additionally, along with the next state and output values, they store extra control signals at each memory location that helps reduce the complexity of controlling the multiplexer. Similar to Garcia-Vargas et al., we also try to reduce the memory size by exploiting the fact that only a subset of inputs are active at

each state. However, we further optimize the memory size by introducing an input encoder which exploits the fact that not all the combinations of the active inputs contribute to different choices of next state selection. For example, in a scenario where the maximum number of active inputs at one state is 3, previous works looks at having $8(2^3)$ memory locations for choosing the potential next state for a given state. However, we show that the number of memory locations can be further reduced to the maximum number of reachable next states per state which is normally less than $2^{(\text{number of active inputs})}$. Moreover, our work further reduces the number of states that must be implemented in memory by looking at the characteristics of HLS-generated benchmarks. We show that a hybrid FSM implementation can be used to divide the task of next state calculation between the memory unit and an accumulator unit, resulting in significant reduction in number of states that must be mapped to the memory and consequently reducing the memory size.

Glaser et al. [11] presents a reconfigurable FSM architecture, TR-FSM, to be implemented as part of the ASICs or System On a Chip (SOC) designs. The proposed architecture offers reduced area, delay, and power consumption compared to an FPGA baseline. However, their architecture must be sized according to the specifications of the FSMs that are going to be implemented onto this architecture, otherwise the extra resources will be wasted. TR-FSM is possible in case of ASIC and SOC design for a certain class of applications where they can profile FSMs prior to generating the TR-FSM block. However, their work cannot be utilized as a general architecture where the size of FSMs is not known in advance, limiting the feasibility for integrating their proposed architecture into the FPGA architecture. However, in our work, we are able to design specialized FSM blocks that can benefit the common FSM sizes, while still allowing the mapping of larger FSMs using the proposed fracturable architecture.

Wilson et al. [35] propose a low overhead FSM overlay based on a multi-RAM architecture. They aim to improve the area usage of the previously proposed memory-based overlays by grouping the state machines to different subsets based on the number of active inputs at each state. The states at each subset are then mapped to separate memory units such that each memory address space can be tailored to the number of active inputs in each subset. Their solution, however, still has a larger area compared to the LUT implementation, since the main goal of their

work is to reduce the FPGA compilation time.

# Chapter 7

# Future Work

In this dissertation, we explore the potential of adding a specialized hard block for FSM implementation to existing FPGA architectures to reduce the area of the next state calculation logic in FSMs that are generated by HLS tools. In the future, we plan to explore the possibility of extending the proposed architecture to also improve the area efficiency of the output calculation part of FSMs.

Additionally, we plan to fully integrate our specialized block to a baseline FPGA architecture. To realize this goal, we also need to extend our mapping flow to support the full CAD flow that can compile the FSM onto the modelled FPGA architecture. The full CAD flow requires support for technology mapping, placement, and routing of the applications onto the modified FPGA architecture.

Another future direction to further optimize our proposed architecture is to explore the potentials of adding hard routing between the fracturable FSM blocks to improve the area and delay for the large FSMs that need to be mapped to a combination of fracturable FSM blocks. In order to support the modified routing structure, the placement and routing algorithms need to be modified accordingly to maximize the efficiency of hard routing between the fracturable blocks.

Our work shows that there are potentials for introducing new hard blocks to the existing FPGA architectures due to the adoption of HLS tools among hardware designers. The recent adoption of FPGAs in new domains, such as cloud computing, also suggests a future direction to explore the potentials of adding new hard blocks to the current FPGA architecture that are well suited for accelerating the common

operations in these new domains. One potential solution to approach this problem is by using graph isomorphism algorithms to find the common structure and logic operations in this new domains which can be used as a starting point to design new hard blocks to be integrated to FPGAs.

# Chapter 8

# Conclusion

In this dissertation, we analyzed the control-unit portion of RTL designs that are generated by HLS tools. HLS-generated control units, modelled by finite-state machines, often have a large influence on the total area of the design in applications where data-path realization requires a large number of states and control signals. We show that these FSMs demonstrate common properties that can be exploited to improve the area of FSM implementations.

We propose a novel mix-grained architecture that takes advantage of these characteristics to improve the total area for implementing the next state calculation logic in FSMs. The proposed architecture can be integrated to modern FPGA architectures. We introduce a new state assignment technique that enables FSMs to better map to our proposed architecture. We evaluate our proposed architecture on a group of RTL designs generated by a commercial HLS tool. Finally, we show that the proposed architecture is on average 3X smaller than LUT-based FSM implementations on a baseline FPGA. The reduction in area is achieved without affecting the performance of the design.

# Bibliography


[1] Altera. FPGAs for High-Performance DSP Applications. https://www.altera.com/en_US/pdfs/literature/wp/wp_dsp_comp.pdf, . → pages 74

[2] Altera. Floating-point DSP Energy Efficiency on Altera 28 m, FPGAs. https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/wp/wp-01192-bdti-altera-fp-dsp-energy-efficiency.pdf, . → pages 74

[3] Amazon. Amazon EC2 F1 Instances-Run Customizable FPGAs in the AWS Cloud. https://aws.amazon.com/ec2/instance-types/f1/. → pages 2

[4] Apache Software Foundation. LuCy. http://lucy.apache.org/, 2016. URL http://lucy.apache.org/. [Online; accessed 19-May-2016]. → pages 51

[5] ArmDeveloper. Artisan Memory Compiler. https://developer.arm.com/products/physical-ip/memory-compilers. → pages 55

[6] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, et al. A cloud-scale acceleration architecture. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–13. IEEE, 2016. → pages 2

[7] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-level synthesis for fpgas: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, 2011. → pages 2, 11, 12

[8] R. H. Dennard, F. H. Gaensslen, and K. Mai. Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions. In *IEEE Journal of Solid-State Circuits*, October 1974. → pages 1

[9] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Papers on Twenty-five years of electronic design automation*, pages 241–247. ACM, 1988. → pages 29

[10] I. Garcia-Vargas, R. Senhadji-Navarro, G. Jimenez-Moreno, A. Civit-Balcells, and P. Guerra-Gutierrez. Rom-based finite state machine implementation in low cost fpgas. In *Industrial Electronics, 2007. ISIE 2007. IEEE International Symposium on*, pages 2342–2347. IEEE, 2007. → pages 10, 74

[11] J. Glaser, M. Damm, J. Haase, and C. Grimm. Tr-fsm: transition-based reconfigurable finite state machine. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 4(3):23, 2011. → pages 75

[12] Hwaci. SQLite. https://www.sqlite.org, 2016. URL https://www.sqlite.org. [Online; accessed 02-June-2016]. → pages 51

[13] Intel. Intel Completes Acquisition of Altera. https://newsroom.intel.com/news-releases/intel-completes-acquisition-of-altera/. → pages 2

[14] I. Kuon and J. Rose. Measuring the gap between fpgas and asics. *IEEE transactions on computer-aided design of integrated circuits and systems*, 26 (2):203–215, 2007. → pages 3, 6

[15] I. Kuon, R. Tessier, and J. Rose. Fpga architecture: Survey and challenges. *Foundations and Trends in Electronic Design Automation*, 2(2):135–253, 2008. → pages x, 6, 7

[16] M. Langhammer and B. Pasca. Floating-point dsp block architecture for fpgas. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 117–125. ACM, 2015. → pages 74

[17] M. Langhammer and B. Pasca. Floating-point dsp block architecture for fpgas. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 117–125. ACM, 2015. → pages 3

[18] C. Lo and P. Chow. Model-based optimization of high level synthesis directives. In *Field Programmable Logic and Applications (FPL), 2016 26th International Conference on*, pages 1–10. IEEE, 2016. → pages 17, 52

[19] J. Luu, J. Goeders, M. Wainberg, A. Somerville, T. Yu, K. Nasartschuk, M. Nasr, S. Wang, T. Liu, N. Ahmed, et al. Vtr 7.0: Next generation architecture and cad system for fpgas. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 7(2):6, 2014. → pages 56, 57

[20] C. Menn, O. Bringmann, and W. Rosenstiel. Controller estimation for fpga target architectures during high-level synthesis. In *Proceedings of the 15th international symposium on System Synthesis*, pages 56–61. ACM, 2002. → pages 3

[21] G. D. Micheli. *Synthesis and optimization of digital circuits*. McGraw-Hill Higher Education, 1994. → pages 7, 8, 9, 18, 29, 31

[22] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, 1965. → pages 1

[23] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 13–24. IEEE, 2014. → pages 2

[24] R. David Evans. Architecture Synthesis from High-Level Specifications. → pages 51

[25] B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei, and D. Brooks. Machsuite: Benchmarks for accelerator design and customized architectures. In *Workload Characterization (IISWC), 2014 IEEE International Symposium on*, pages 110–119. IEEE, 2014. → pages 51

[26] R. Senhadji-Navarro, I. Garcia-Vargas, and J. L. Guisado. Performance evaluation of ram-based implementation of finite state machines in fpgas. In *Electronics, Circuits and Systems (ICECS), 2012 19th IEEE International Conference on*, pages 225–228. IEEE, 2012. → pages 10

[27] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli. Sis: A system for sequential circuit synthesis. 1992. → pages 54

[28] Y. S. Shao and D. Brooks. Research infrastructures for hardware accelerators. *Synthesis Lectures on Computer Architecture*, 10(4):1–99, 2015. → pages 1, 2

[29] Y. Shi, C. W. Ting, B.-H. Gwee, and Y. Ren. A highly efficient method for extracting fsms from flattened gate-level netlist. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, pages 2610–2613. IEEE, 2010. → pages 54

[30] V. Sklyarov. Synthesis and implementation of ram-based finite state machines in fpgas. *Field-Programmable Logic and Applications: The Roadmap to Reconfigurable Computing*, pages 718–727, 2000. → pages 10

[31] V. Sklyarov. An evolutionary algorithm for the synthesis of ram-based fsms. In *International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems*, pages 108–118. Springer, 2002. → pages 10

[32] SPEC. CPU 2006. https://www.spec.org/cpu2006/. → pages 19, 51

[33] Synopsys. Design Compiler. http://www.synopsys.com/Tools/ Implementation/RTLSynthesis/ DesignCompiler/Pages/default.aspx. → pages 55

[34] A. Tiwari and K. A. Tomko. Saving power by mapping finite-state machines into embedded memory blocks in fpgas. In *Proceedings of the conference on Design, automation and test in Europe-Volume 2*, page 20916. IEEE Computer Society, 2004. → pages 10

[35] D. Wilson and G. Stitt. A scalable, low-overhead finite-state machine overlay for rapid fpga application development. *arXiv preprint arXiv:1705.02732*, 2017. → pages 75

[36] S. J. E. Wilton, J. Rose, and Z. Vranesic. Architectures and algorithms for field-programmable gate arrays with embedded memory. *University of Toronto, Toronto, Ont., Canada*, 1997. → pages 74

[37] C. Wolf. Yosys open synthesis suite, 2015. → pages 53

[38] Xilinx. Xilinx and IBM to Enable FPGA-Based Acceleration within SuperVessel OpenPOWER Development Cloud. https://www.xilinx.com/news/press/2016/xilinx-and-ibm-to-enable-fpga-based-acceleration-within-supervessel-openpower-development-cloud.html, . → pages 2

[39] Xilinx. Qualcomm and Xilinx Collaborate to Deliver Industry-Leading Heterogeneous Computing Solutions for Data Centers with New Levels of Efficiency and Performance. https://www.xilinx.com/news/press/2015/qualcomm-and-xilinx-collaborate-to-deliver-industry-leading-heterogeneous-computing-solutions-for-data-centers-with-new-levels-of-efficiency-and-performance.html, . → pages 2

[40] Xilinx. High-Volume Spartan-6 FPGAs: Performance and Power Leadership by Design. https://www.xilinx.com/support/documentation/white_papers/wp396_S6_HV_Perf_Power.pdf, . → pages 74

[41] Xilinx. Spartan-7 FPGAs: Meeting the cost-Sensitive Market Requiremnts. https://www.xilinx.com/support/documentation/white.../wp483-spartan-7-intro.pdf, . → pages 74

[42] Xilinx. Vivado High-Level Synthesis. http://www.xilinx.com/products/design-to ols/vivado/, . → pages 2, 16

[43] Xilinx. Embedded Vision with INT8 Optimization on Xilinx Devices. https://www.xilinx.com/support/documentation/white_papers/wp490-embedded-vision-int8.pdf, . → pages 74

[44] Xilinx-Xcell Daily Blog. Baidu Adopts Xilinx Kintex UltraScale FPGAs to Accelerate Machine Learning Applications in the Data Center. https://forums.xilinx.com/t5/Xcell-Daily-Blog/Baidu-Adopts-Xilinx-Kintex-UltraScale-FPGAs-to-Accelerate/ba-p/728667. → pages 2

[45] G. Zgheib, L. Yang, Z. Huang, D. Novo, H. Parandeh-Afshar, H. Yang, and P. Ienne. Revisiting and-inverter cones. In *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, pages 45–54. ACM, 2014. → pages 56