Summer 9-7-2023

Feature-Oriented Hardware Design

Justin Deters
Washington University – McKelvey School of Engineering

Follow this and additional works at: https://openscholarship.wustl.edu/eng_etds

Recommended Citation
Deters, Justin, "Feature-Oriented Hardware Design" (2023). McKelvey School of Engineering Theses & Dissertations. 952.
https://openscholarship.wustl.edu/eng_etds/952

This Dissertation is brought to you for free and open access by the McKelvey School of Engineering at Washington University Open Scholarship. It has been accepted for inclusion in McKelvey School of Engineering Theses & Dissertations by an authorized administrator of Washington University Open Scholarship. For more information, please contact digital@wumail.wustl.edu.
WASHINGTON UNIVERSITY IN ST. LOUIS

McKelvey School of Engineering
Department of Computer Science & Engineering

Dissertation Examination Committee:
Ron Cytron, Chair
Jeremy Buhler
Roger Chamberlain
Christopher Gill
Krishna Kavi

Feature-Oriented Hardware Design
by
Justin Deters

A dissertation presented to
the McKelvey School of Engineering
of Washington University in
partial fulfillment of the
requirements for the degree
of Doctor of Philosophy

August 2023
St. Louis, Missouri
# Table of Contents

List of Figures ................................................................. v
Acknowledgments ............................................................. ix
Abstract ................................................................................... xi

## Chapter 1: Introduction ......................................................... 1
  1.1 Hardware Evolution ....................................................... 1
  1.2 Hardware Landscape ..................................................... 2
  1.3 How do we get there? ..................................................... 4

## Chapter 2: Related Work ...................................................... 7
  2.1 Feature-Oriented Programming .......................................... 7
  2.2 Hardware Generation Languages ....................................... 9
  2.3 Monolithic Designs ....................................................... 10
  2.4 Current Chisel Tools for Feature-Orientation ....................... 11
    2.4.1 Diplomacy ......................................................... 11
    2.4.2 Chisel Aspects .................................................... 13

## Chapter 3: Adaptive Adder .................................................. 16
  3.1 Introduction ........................................................... 16
  3.2 Design ............................................................... 17
  3.3 Results ............................................................... 18

## Chapter 4: Feature Application in the Elaboration Process .......... 21
  4.1 Lexical Application ...................................................... 21
  4.2 Semantic Application .................................................. 24
  4.3 Emitter Application .................................................... 26

## Chapter 5: Feature-Oriented Performance Counter Designs .......... 27
  5.1 Introduction ........................................................... 27
    5.1.1 Research Questions ............................................. 28
  5.2 Current Monolithic Design ............................................ 28
  5.3 Feature Application using Scala Trees ............................... 29
    5.3.1 Scala Trees ..................................................... 30
List of Figures

Figure 1.1: Design endpoints on the qualitative and quantitative spectrum of hardware characterization. .................................................. 2

Figure 2.1: AspectJ example of logging from the AspectJ Programming Guide. [28] .............................................................. 8

Figure 2.2: A very small Verilog example. ...................................................... 9

Figure 2.3: The tool chain we are using for hardware generation. .......... 10

Figure 2.4: An example ALU from RISC-V-mini ................................ 12

Figure 2.5: The basic adder code for Diplomacy. ................................ 14

Figure 2.6: The classes that drive the adder with different parameters to be negotiated. 15

Figure 3.1: Estimated potential bugs. ...................................................... 16

Figure 3.2: An $S(16, 2)$ Adder. This adder is 16 bits long and has 2 sub-adders each of length 8. ..................................................... 18

Figure 3.3: The number of lines of code to generate the hybrid designs. 19

Figure 3.4: Lines of code generated per generator line of code. .......... 19

Figure 3.5: The number of LUTs used in the hybrid design. ................. 20

Figure 3.6: The maximum delay for the hybrid design. ....................... 20

Figure 4.1: The levels of design elaboration in Chisel execution. .......... 21

Figure 4.2: Code for creating the structure of RISC-V-mini. ............... 22

Figure 4.3: A simple ALU with hooks. .................................................. 23
Figure 4.4: A small Chisel FSM example [68].

Figure 5.1: A feature unit for implementing the base of the counter system.

Figure 5.2: A more modular CSR structure.

Figure 5.3: The dependency relationships between the features.

Figure 5.4: Space of our endpoints in LUTs.

Figure 6.1: FSM for a vending machine that accepts 5 cent coins and dispenses peanuts that cost 10 cents.

Figure 6.2: Dependencies between vending machine features.

Figure 6.3: Dependencies between Nim features.

Figure 6.4: FSM for a 5-token heap that allows one or two tokens to be removed in a turn.

Figure 6.5: FSM specifying alternation of Players A and B.

Figure 6.6: Nim Finite-State Machine.

Figure 6.7: The resulting FSM of the application of Print Funds to the FSM from Figure 6.1.

Figure 6.8: Examples of before and after advice on a small FSM.

Figure 6.9: The resulting FSM of the application of Peanut Warning to the FSM from Figure 6.1.

Figure 6.10: The implementation of the Print Funds feature in Foam.

Figure 6.11: Construction of the Cross Product FSM. The inputs A and B must be deterministic; the output C is deterministic.

Figure 6.12: Number of generated states, transitions, and LUTs depending on selected features.

Figure 6.13: Number of generated states and transitions by selected features for traditional Nim.
Figure 6.14: Number of generated states and transitions by selected features for the Subtraction game. ................................................................. 60
Figure 6.15: Number of generated states and transitions by selected features for Circle Nim. ................................................................. 61
Figure 7.1: A pointcut from our implementation. ................................. 64
Figure 7.2: Advice using the pointcut from Figure 7.1 from our implementation. ... 64
Figure 7.3: Using an FSM handle in our software library. ....................... 65
Figure 7.4: The Read FSM. ................................................................. 67
Figure 7.5: The FSM that combines Read, Write, and Acknowledge Idle. ...... 68
Figure 7.6: The FSM that combines Read, Write, Acknowledge Read, and Dirty Accounting. ................................................................. 69
Figure 7.7: Endpoints for the Instruction and Data Caches. ...................... 70
Figure 7.8: Lines of code used to implement each feature. ...................... 71
Figure 7.9: The implementation of the Cache class. ............................. 76
Figure 7.10: HasMiddleAllocate feature. ............................................. 77
Figure 7.11: Dusty feature in Faust. ..................................................... 77
Figure 7.12: The cache FSM from the RISC-V Mini. .............................. 78
Figure 7.13: The cache FSM from RISC-V Mini with the MSI protocol. ...... 79
Figure 7.14: Cycles per instruction for each RISC-V benchmark, by endpoint. 80
Figure 7.15: The area in LUTs of the endpoints. ................................. 81
Figure 8.1: A simple cache hit event in Faust 2. .................................... 83
Figure 8.2: An aspect that aggregates events and creates performance counters. ... 83
Figure 8.3: Re-implementation of the performance counter system. .......... 85
Figure A.1: The base FSM for the vending machine. ............................ 94
Figure A.2: The type signature of the Add Coin class. ............................. 95
Figure A.3: The predicate for the Add Coin class. ............................... 96
Figure A.4: Function signature for the implementation function for advice. ... 96
Figure A.5: The full \texttt{AddCoin} class. ........................................ 97
Figure A.6: Advice used by Peanut Warning to add a new warning to the FSM. . 99
I would like to acknowledge past educators in my life: Rich Ingmire, Bob McNeill, Bonnie Staudt, Carol DeFreese, Robert Cliffe, Dr. Ruthie Dare-Halma, Dr. Michael Rudy, and Dr. Brandon Bolte. Each in their way contributed to pushing me to the path of the PhD. I would like to thank Clayton Faber for spending many hours with me over the last three years teaching me computer engineering from scratch. I would like to acknowledge Dr. Michela Becchi for giving me my first undergraduate research opportunity. To my fellow members of the Stream Based Supercomputing group, I thank you for your encouragement and input into my research. I especially thank Roger Chamberlain and Jeremey Buhler for guiding and providing the funding for this work. I would like to credit both Max Camp-Oberhauser and Peyton Gozon for their collaboration on Chapter 6. Finally, I would like to thank my dissertation advisor, Dr. Ron Cytron for providing me with the opportunity to explore this research and for teaching me how to fly solo.

Supported under NSF CISE award CNS-1763503 Performant Architecturally Diverse Systems via Aspect-oriented Programming.

Justin Deters

Washington University in St. Louis
August 2023
In memory of John Berchin.
The sea of hardware features is vast. Hardware features range from things as simple and small as a bypass line in the datapath of a processor to as large and complicated as a cache or accelerator. Qualitatively, hardware designers choose what functionality the design will have through the instruction set architecture (ISA). Will the processor support floating-point instructions? Will the processor’s applications require SIMD or cryptography instructions? These considerations lie on a spectrum from limited or specialized ISAs to full featured general processing ISAs. This spectrum intersects with the quantitative spectrum of the microarchitecture. A processor must reach the die area, electrical power, and performance targets in its design. Generally, this ranges from small, low powered, and slow processors to large, power hungry, and fast processors. Should the processor have 4 or 8 cores? How deep should the datapath be? How many levels of cache should the processor contain? To build a hardware characterization, designers carefully select different features to satisfy functional requirements while attempting to satisfy space, power, and performance constraints along the way.

We envision a hardware characterization that can evolve both qualitatively and quantitatively. In a feature-oriented characterization of an ISA, say RISC-V, hardware designers could easily swap their features for others. This could open up a marketplace of hardware features where designers could test and create features collaboratively. With a common
ISA and feature-oriented characterization, researchers could better understand the nature of hardware components by reducing the variability between different hardware implementations. Instead of starting from scratch, the characterizations could build upon already established features.

In this dissertation we work towards this goal of feature-oriented hardware characterization by exploring novel hardware characterization techniques through feature-oriented programming. This research is divided into four main thrusts.

- Modifying the generated logic to incorporate new features.
- Modifying the control structures to incorporate new features.
- Exploring types to make hardware designs compatible with feature application.
- Specifically addressing cross-cutting features of hardware characterizations.
Chapter 1

Introduction

The sea of hardware features is vast. Hardware features range from things as simple and small as a bypass line in the datapath of a processor to as large and complicated as a cache or accelerator. Qualitatively, hardware designers choose what functionality the design will have through the instruction set architecture (ISA). Will the processor support floating-point instructions? Will the processor’s applications require SIMD or cryptography instructions? These considerations lie on a spectrum from limited or specialized ISAs to full featured general processing ISAs. This spectrum intersects with the quantitative spectrum of the microarchitecture, illustrated in Figure 1.1. A processor must reach the die area, electrical power, and performance targets in its design. Generally, this ranges from small, low powered, and slow processors to large, power hungry, and fast processors. Should the processor have 4 or 8 cores? How deep should the datapath be? How many levels of cache should the processor contain? To build a hardware characterization, designers carefully select different features to satisfy functional requirements while attempting to satisfy space, power, and performance constraints along the way.

1.1 Hardware Evolution

To achieve price points, a single ISA may have many numerous characterizations, each trying to target a particular place on the spectrum. As technology evolves, these characterizations become obsolete and must be re-evaluated and re-designed. Indeed, humanity has come a long way since the first handheld calculators and the Intel 4004. With each advancement of technique and technology, hardware designs must evolve to support them. The Intel 4004 had a word size of only 4 bits [15]. Since then, technology has progressed to the common 64 bit word size. Each jump in word size precipitated a redesign that would affect the entire chip. In addition, as hardware evolves so must the control structures. Since the IBM Stretch’s...
2-stage pipeline [14], processor pipelines have become far deeper, with tens of stages. Adding just one more stage requires evolving the control structures that cross-cut the entire pipeline.

A prolific example of this evolution, that has maintained backwards compatibility over 45 years\(^1\), is the x86 architecture. Over that time, the x86 architecture has survived the jump between 16, 32, and 64 bits, the addition of pipelining, superscalar, speculative execution, and simultaneous multithreading, and numerous new instructions for SIMD, transactional memory, and virtualization [5, 72, 27, 62, 55].

1.2 Hardware Landscape

Today, the hardware characterization landscape is more diverse than ever with a huge range of architectural and microarchitectural features. Managing all these features to create a cohesive design is an immense undertaking. Gone are the days of the literal “tape-out” of

\(^1\)We must exclude the IA-64 architecture. This architecture was not backwards compatible with x86 and only had one generation.
circuit designs. Instead many teams of people work collectively with computer aided design tools to make a finished chip. Consider the following examples of modern hardware features.

Caches  Beginning as simple units that sit just outside the datapath, caches have evolved to cross-cut an entire chip design. Instead of a single L3 cache, the Intel Cascade Lake microarchitecture [20] utilizes a split design. Each core is partnered with a section of the L3 cache on a “core tile”. The tiles are arranged in a row-column pattern with a mesh interconnect to facilitate communications between sections of the cache. This product line ranges from 4 to 28 cores. Managing different design variations within this product line by hand would be difficult and tedious. Changing the interconnect fabric cross-cuts the entire chip design, requiring manual refactoring across the whole of the design. The designers could decide to replace some cores with high efficiency ones to meet power constraints. Determining the proper ratio of high performance to efficiency cores at each core count requires numerous design iterations. The layout of the cores could also affect performance across the interconnect.

Pipeline  Moving to the datapath, since the introduction of the Zen microarchitecture, AMD processors have had 19-stage pipelines [79]. Forwarding and bypassing affects many places within a large pipeline like this. In addition, some features, such as multipliers and floating point units, cross-cut stages of the pipeline and are pipelined themselves. This is further complicated by the fact that Zen is a superscalar microarchitecture. All forwarding and bypassing must account for pipeline hazards across many instructions simultaneously. Increasing instruction level parallelism could have consequences for the whole pipeline.

Heterogeneous Processors  In our post-Moore world, heterogeneous computing has reached consumer-grade hardware. The Apple M2 [6] line of Arm processors features high-efficiency cores, performance cores, a graphics processing unit, specialty hardware for machine learning, image processing, and a security core. Managing the design space complexity of heterogeneous hardware cross-cuts the design. What instructions must the ISA include to support the accelerator hardware? Should each core get its own small accelerator, or should cores share a single accelerator? Should the CPU and GPU share cache or each have their own? All these questions must be answered in the design space exploration.
1.3 How do we get there?

In their 2017 Turning Award lecture, John Hennessy and David Patterson [35] outlined a vision for an open and modular ISA in the form of RISC-V. RISC-V applications span small, application-specific processors (such as the controllers for Western Digital storage products [78] and Google’s Titan M2 security chip [47]) to high-performance processors (such as the SiFive P550 [70]). RISC-V [75, 76] itself is divided into modular components that allow for customization to specific applications. The base RISC-V specification includes only integer instructions. If necessary, one may add the optional floating-point instructions. But, practically, what does this take? New hardware needs to be added to decode the instructions, a floating point unit must be installed into the datapath, new control structures must be included, and much more. Moreover, floating point instructions are not expected to complete in a single cycle.

Furthermore, hardware designers must carefully choose which implementations of features to include in their designs. Features themselves also exist within the same design spectrum as the overall characterization. There is always a trade off between performance, power consumption, and circuit area. Within the RISC-V Specification [75, 76] designers can choose from single, double, and quad precision floating point instructions. With each increase in precision, more power and area are required to accommodate. Normally, replacing one level of precision with another would be a very tedious process, requiring large changes in many places in the hardware design. Designers must also consider the trade-offs between different microarchitectural implementations of the floating point unit, especially in terms of pipeline stages as the FPU is typically pipelined itself.

Unfortunately, despite the modularity of RISC-V, it still ended up with many different and divergent characterizations (Section 2.3). Although these all implement some form of RISC-V, the underlying characterization of features differs between all of these.

The Linux kernel long ago solved the problem of adaptation to different requirements. There is a single mainline Linux kernel, but it is altered to suit different applications through modularity. Over the past 31 years, Linux has been adapted to numerous ISAs, developments in hardware, and has thousands of different flavors through distributions. We posit that hardware characterizations should take a note from the software community. That is, the
hardware characterizations should be *feature-oriented*. Rather than creating characterizations that are tied to a specific version of an architecture and microarchitecture, it should adapt to specific use cases and evolve over time to include new hardware features. Hardware could benefit from feature-oriented design in the following ways:

- Simpler modification of logic to incorporate new features into a design.
- Higher leverage of code to produce designs automatically.
- Enabling a larger design space to explore between different hardware characterizations.

We envision a hardware characterization that can evolve both qualitatively and quantitatively. In a feature-oriented characterization of an ISA, say RISC-V, hardware designers could easily swap their features for others. This could open up a marketplace of hardware features where designers could test and create features collaboratively. With a common ISA and feature-oriented characterization, researchers could better understand the nature of hardware components by reducing the variability between different hardware implementations. Instead of starting from scratch, the characterizations could build upon already established features.

In this dissertation we work towards this goal of feature-oriented hardware characterization by exploring novel hardware characterization techniques through feature-oriented programming. This research is divided into four main thrusts.

- Modifying the generated logic to incorporate new features.
- Modifying the control structures to incorporate new features.
- Exploring types to make hardware designs compatible with feature application.
- Specifically addressing cross-cutting features of hardware characterizations.

The organization of this dissertation is as follows. In Chapter 2 we cover related work and context for the dissertation. Chapter 3 introduces a proof-of-concept for feature-oriented hardware in the form of an adaptive adder [23]. Chapter 4 discusses the levels within the Chisel toolchain where features are applied. The main thrust of the research questions are
covered in Chapters 5, 6, and 7 where we cover our feature-oriented performance counter system [25], feature-oriented framework for finite-state machines, and our feature-oriented cache [24], respectively. Finally, in Chapter 8 we propose future work after this dissertation.


Chapter 2

Related Work

2.1 Feature-Oriented Programming

In large software systems, developers traditionally focus on nouns and verbs of the system. This decomposition results in the objects and methods, respectfully. However, some software engineers choose to use a feature-oriented approach [1]. Instead of a featureful implementation, which must include all the infrastructure for every feature that any end user would use, feature-oriented programming focuses on creating a smaller base system to which end users can apply features. Often, this requires refactoring of the codebase to be amenable to feature application.

Aspect-Oriented Programming  We use Aspect-Oriented Programming (AOP) [44] to implement our features. AOP developed as a way to capture crosscutting concerns in software systems. The canonical example of this is logging. Typically, logging code is inserted by hand at every point in the system. AOP allows programmers to capture the implementation in what is called an aspect. An aspect contains a pointcut and advice. A pointcut describes a set of code locations (called join points) where implementation information will be inserted. The implementation information captured in the advice is then “woven” into the code before, after, or around all the join points in the pointcut. For logging, one could capture every instance of a particular function in a pointcut and designate that logging code to be inserted after each call to that function.

The most mature example of this programming paradigm is AspectJ [29]. This language extends the Java language with aspect-oriented constructs. Prior work has demonstrated the utility of using AspectJ to implement features within a Java program. Specifically, [39, 40, 38, 52] in the context of the CORBA [56] Event Channel, which is responsible for
aspect Logging {
    OutputStream logStream = System.err;

    pointcut move():
        call(void FigureElement.setXY(int, int)) ||
        call(void Point.setX(int)) ||
        call(void Point.setY(int)) ||
        call(void Line.setP1(Point)) ||
        call(void Line.setP2(Point));

    before(): move() {
        logStream.println("about to move");
    }
}

Figure 2.1: AspectJ example of logging from the AspectJ Programming Guide. [28]

publishing events to subscribers. The nature and disposition of the publishers and subscribers dictate the features needed in the Event Channel. In its monolithic form, unnecessary features avoid execution, but their presence still consumes memory. Moreover, even a simple runtime check concerning a feature’s inclusion can adversely affect latency for time-critical applications.

Instead of removing features they did not want, they took a compositional approach. By stripping down the system and then adding features only when needed, they were able to get significant savings in memory area and increases in performance.

A feature-specific Event Channel was evaluated and compared with a monolithic implementation in terms of area and throughput [52]. For area, the feature-specific version was 1.4–3 times smaller than the monolithic version: an appreciable savings for an embedded systems application. The throughput for the feature-specific version was approximately 131,000 events per second, compared to approximately 1,600 events per second in the monolithic version, with unnecessary features present but disabled.

The common pattern of inserting if-else statements into a code base to control features clutters the codebase with unused features, obscuring the control flow of the program. Furthermore, upon execution each of these statements must be evaluated, so a large number of features can reduce performance. In particular, Lohmann et al. showed a significant reduction in memory footprint while achieving a higher level of separation of concerns in their embedded system software [51] and Zhang et al. demonstrated the use of AOP in a
module OddOrEven(
    input  clock,
    input  reset,
    input [7:0] io_in,
    output io_out);

wire [7:0] _T_1 = 8'h1 + io_in;
assign io_out = !_T_1[0];
endmodule

Figure 2.2: A very small Verilog example.

middleware system by achieving an 8% performance improvement over their original implementation with a 40% reduction in code size [81].

2.2 Hardware Generation Languages

Most hardware designers still write their designs in Verilog [10] or VHDL [12]. These hardware description languages are relatively low-level compared to their software counterparts. Although they do provide some abstractions, such as the Module construct, they operate relatively the same level as software assembly languages.

Hardware generation languages (HGL), such as [11, 8, 73, 18, 2, 22, 50], have made progress in lifting hardware design to higher level languages. Many of these languages are embedded within a high-level programming language and leverage the provided constructs in organizing and abstracting hardware. Rather than directly describing the resulting hardware in the language itself, HGLs produce a design as a result of executing a program. The most prolific of this family of languages is Chisel [11].

Chisel is an HGL embedded in Scala [57]. As such, it contains all of the object-oriented and functional features of the Scala language. A Chisel program is a combination of Scala constructs and custom hardware constructs that generate hardware depending on the program execution. As a Chisel program executes the hardware constructs, an internal representation of the circuit is built in a global data structure [48]. The resulting in-memory representation of the circuit is then emitted into an intermediate circuit representation called FIRRTL [43].
The FIRRTL compiler then performs several lowering and optimization passes (such as constant propagation and deduplication [43]). From there the circuit can then be emitted into Verilog or it can be compiled into simulator code using Verilator [74]. Figure 2.3 shows the Chisel toolchain from Scala to hardware synthesis.

![Chisel Toolchain](image)

**Figure 2.3:** The tool chain we are using for hardware generation.

### 2.3 Monolithic Designs

Despite the introduction of HGLs such as Chisel, hardware designers tend to use the same monolithic design methodology borrowed from Verilog [10] and VHDL [12]. Examples of hardware generators that follow this pattern include:

- **RISC-V-mini** [45] is an introductory hardware generator for Rocket Chip. It implements an in-order three stage pipeline. Being so simple, this generator is useful for pedagogy.

- **Rocket Chip** [9] is a fully featured in-order five-stage pipeline. Rocket Chip is designed to be production ready and for end users to customize it according to their needs.

- **BOOM** [16] borrows much of the infrastructure from Rocket Chip, but is instead an out-of-order implementation.

These projects tend to be organized even more restrictively than other software systems. Consider the example ALU in Figure 2.4. This is a typical example of how the above hardware generators are organized. Chisel does require that all hardware constructs be called in the execution path the constructor of a class that extends `Module`. As long as this rule is followed, any other modularity and constructs of the software can be used. Despite
this, hard-coding all hardware into large run-on constructors is the norm. Functions and other classes are rarely used. This leads to many hardware features being entangled and obscured within the structure of the hardware generator. Extending the functionality of these generators can become very tedious and difficult. As a result of this style of software organization, we have three separate diverging characterizations for the same ISA all written in Chisel. This should be a natural environment to swap hardware features, but on the microarchitectural level, these characterizations are not compatible. RISC-V-mini cannot use BOOM’s out-of-order pipeline, Rocket Chip cannot use RISC-V-mini’s cache, etc.

2.4 Current Chisel Tools for Feature-Orientation

The Chisel community has developed two tools that could be used for feature-orientation of hardware. While neither was developed as such, they have the ability to perform some feature-oriented design.

2.4.1 Diplomacy

Diplomacy [19] negotiates parameters between different hardware generators. This tool is primarily used within the Rocket Chip generator for their TileLink [19] chip interconnect. For example, Diplomacy will ensure that the width of every input and output wire agrees across the collection of generators being connected. In this way, Diplomacy manages cross-cutting features. However, from a usability standpoint, Diplomacy does not fair any better than just if-else statements. All hardware modules that use Diplomacy for parameter negotiation must split their execution into two phases. The first negotiates the parameters and then the actual implementation will be executed. Furthermore, each of these classes must be manually prepared for use with Diplomacy by breaking the implementation into two separate classes: one for the Diplomacy framework and a separate class that is called from the Diplomacy class with the module implementation. This obscures the decomposition of the hardware generator as well as requires a lot of manual refactoring to use Diplomacy correctly.
```scala
class AluArea(val width: Int) extends Alu {
  val io = IO(new AluIO(width))
  val cmp =
    Mux(io.A(width - 1) === io.B(width - 1), sum(width - 1), Mux(io.alu_op
    (1), io.B(width - 1), io.A(width - 1)))
  val shamt = io.B(4, 0).asUInt
  val shin = Mux(io.alu_op(3), io.A, Reverse(io.A))
  val shiftr = (Cat(io.alu_op(0) && shin(width - 1), shin).asSInt >> shamt
    )(width - 1, 0)
  val shiftl = Reverse(shiftr)

  val out =
    Mux(
      io.alu_op === ALU_ADD || io.alu_op === ALU_SUB, sum,
      Mux(
        io.alu_op === ALU_SLT || io.alu_op === ALU_SLTU, cmp,
        Mux(
          io.alu_op === ALU_SRA || io.alu_op === ALU_SRL, shiftr,
          Mux(
            io.alu_op === ALU_SLL, shiftl,
            Mux(
              io.alu_op === ALU_AND, io.A & io.B,
              Mux(
                io.alu_op === ALU_OR, io.A | io.B,
                Mux(io.alu_op === ALU_XOR, io.A ^ io.B, Mux(io.alu_op ===
              )
          )
        )
      )
    )

  io.out := out
  io.sum := sum
}
```

Figure 2.4: An example ALU from RISC-V-mini
Figures 2.5 and 2.6 show the simple adder from the Diplomacy tutorial in the Rocket Chip repository [9]. Here, two components drive a one-bit adder and a component to check the result of the adder. Diplomacy negotiates the widths of the input and output lines to ensure that all components on the interconnect are properly parameterized for connection. Hypothetically, Diplomacy could control any feature that could be turned on and off through parameterization, even controlling the configuration of the whole design. But, as the example below shows, this requires a lot of code even for simple designs.

2.4.2 Chisel Aspects

Chisel does include an aspect-oriented library [42]. This library is intended to insert “design collateral” (i.e. secondary features) into Chisel design post-elaboration. We have found that Chisel Aspects lack in very serious ways. The only join point available is the module. Chisel Aspects insert new hardware after elaboration, meaning that no Scala constructs can be joined upon, nor any other hardware constructs. While Chisel Aspects does give access to the rest of the hardware in the module, typical AOP design would allow access to the hardware within the module via the join point context. Instead of providing fine-grained additions to the hardware design, Chisel Aspects are coarse-grained and were not suitable for our feature-oriented methodology.

Chisel Aspects applies features right before emitting FIRRTL. This means that the entire design must be elaborated first. However, it is not strictly necessary to wait for this phase to complete before applying features. In the next chapter, we discuss the different phases of the Chisel elaboration process and how features can be applied in different phases depending on the information they require.
case class UpwardParam(width: Int)
case class DownwardParam(width: Int)
case class EdgeParam(width: Int)

object AdderNodeImp extends SimpleNodeImp[DownwardParam, UpwardParam, EdgeParam, UInt]
{
  def edge(pd: DownwardParam, pu: UpwardParam, p: Parameters, sourceInfo: SourceInfo) = {
    if (pd.width < pu.width) EdgeParam(pd.width) else EdgeParam(pu.width)
  }
  def bundle(e: EdgeParam) = UInt(e.width.W)
  def render(e: EdgeParam) = RenderedEdge("blue", s"width = ${e.width}")
}

class AdderDriverNode(widths: Seq[DownwardParam])(implicit valName: ValName)
  extends SourceNode(AdderNodeImp)(widths)

class AdderMonitorNode(width: UpwardParam)(implicit valName: ValName)
  extends SinkNode(AdderNodeImp)(Seq(width))

class AdderNode(dFn: Seq[DownwardParam] => DownwardParam,
  uFn: Seq[UpwardParam] => UpwardParam)(implicit valName: ValName)
  extends NexusNode(AdderNodeImp)(dFn, uFn)

class Adder(implicit p: Parameters) extends LazyModule {
  val node = new AdderNode {
    { case dps: Seq[DownwardParam] =>
      require(dps.forall(dp => dp.width == dps.head.width), "inward, downward adder widths must be equivalent")
      dps.head
    },
    { case ups: Seq[UpwardParam] =>
      require(ups.forall(up => up.width == ups.head.width), "outward, upward adder widths must be equivalent")
      ups.head
    }
  }
  lazy val module = new LazyModuleImp(this) {
    require(node.in.size >= 2)
    node.out.head._1 := node.in.unzip._1.reduce(_ + _)
  }
  override lazy val desiredName = "Adder"
}

Figure 2.5: The basic adder code for Diplomacy.
class AdderDriver(width: Int, numOutputs: Int)(implicit p: Parameters)
  extends LazyModule {
  // check that node parameters converge after negotiation
  val negotiatedWidths = node.edges.out.map(_.width)
  require(negotiatedWidths.forall(_ == negotiatedWidths.head), "outputs
  must all have agreed on same width")
  val finalWidth = negotiatedWidths.head

  // generate random addend (notice the use of the negotiated width)
  val randomAddend = FibonacciLFSR.maxPeriod(finalWidth)

  // drive signals
  node.out.foreach { case (addend, _) => addend := randomAddend }
}


class AdderMonitor(width: Int, numOperands: Int)(implicit p: Parameters)
  extends LazyModule {
  // print operation
  printf(nodeSeq.map(node => p"${node.in.head._1}").reduce(_ + p" + " + _) + p" = ${nodeSum.in.head._1}"")

  // basic correctness checking
  io.error := nodeSum.in.head._1 != nodeSeq.map(_.in.head._1).reduce(_ + _)
}

Figure 2.6: The classes that drive the adder with different parameters to be negotiated.
Chapter 3

Adaptive Adder

3.1 Introduction

In this section we present a proof-of-concept and motivation for feature-oriented hardware design. Frequently in cryptography there is a need for very large integers to hold cryptographic information. To accelerate ciphering and deciphering, these operations can be transferred to hardware [21, 69, 46, 59]. These designs are very large and complicated. This can lead directly to large amounts of design time spent on debugging.

The book Code Complete estimates that for every 1000 lines of code, 1 - 25 bugs are introduced [53]. Zhang et al. support this claim with their study of several versions of the Eclipse IDE, finding 6 - 11 bugs per 1000 lines of code [82]. Here, we will demonstrate our technique on a 128-bit adder generator capable of producing thousands of lines of code. If we average the number of potential bugs at 12.5 per 1000 lines of code, if produced by hand, Figure 3.1 shows the estimated number of bugs per generation endpoint of our adder design. [77]

Rather than creating these designs by hand, we posit that designers should instead create smaller specifications that can generate correct designs. Smaller specifications reduce opportunities for coding mistakes. Furthermore, by feature-orienting these specifications, a larger

<table>
<thead>
<tr>
<th>Lines of Code</th>
<th>Estimated Bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>7889</td>
<td>98.6</td>
</tr>
<tr>
<td>3291</td>
<td>41.1</td>
</tr>
<tr>
<td>1977</td>
<td>24.7</td>
</tr>
<tr>
<td>1628</td>
<td>20.3</td>
</tr>
<tr>
<td>1579</td>
<td>19.7</td>
</tr>
</tbody>
</table>

Figure 3.1: Estimated potential bugs.
design space can be explored, allowing for better examinations of the trade-offs between space, performance, and power.

3.2 Design

For this preliminary study, we examined the design trade-offs between two classic adder circuits: ripple-carry and carry-lookahead. Ripple-carry adders work much like simple pen-and-paper addition, where the carry moves from one digit to the next. Due to this, these adders have a time complexity of $\theta(n)$. Carry-lookahead adders have circuitry that allows the carry to bypass the lower digits, making it a faster circuit. Instead, these adders have time complexity of $\theta(\log(n))$ at the expense of more space taken up by the extra circuitry. Normally, one of the two types of carries discussed would be hard coded into the hardware design. Hardware synthesis tools for Verilog or VHDL will choose between the two implementations, selecting for minimization of space or maximization of performance.

As discussed in Chapter 1, qualitatively, both circuits perform exactly the same function. However, quantitatively they are single points on the characterization spectrum. Ripple-carry being on the small and slow end, and carry-lookahead being on the fast and large end. Instead, in our approach, we captured each type of carry in an aspect that then can be selectively applied to the design. Thus, we can quickly rearrange modules in the adder and apply carries to them, allowing easy exploration of the optimization space.

As described earlier, Figure 2.3 shows the tool flow of Chisel. The design enters the tool flow on the left fully configured. The Chisel toolchain takes care of generating the circuit in the design and applying the aspects. Verilog of the configured adder is emitted on the other end. Typically, the adder would be generated and optimized at the Vivado (synthesis) stage of the toolchain. With our method, we pull the generation of the adder and its optimization into the language itself at the HGL stage.

To demonstrate this methodology, we created an adder that contains elements from the two adders discussed earlier. Our hybrid designs are $n$ bits long. Each $n$ bits can be divided into $m$ sub-adders that are each $n/m$ bits in length. The sub-adders use carry-lookahead internally, and ripple-carry connects the sub-adders externally. We denote these hybrid adders as $S(n, m)$ adders where $S(16, 1)$ is a classic carry-lookahead adder and $S(16, 16)$ is
a classic ripple-carry adder. For example, Figure 3.2 shows an $S(16, 2)$ adder. The length of the adder is $n = 16$, there are $m = 2$ sub-adders, and then each sub-adder is of length $n/m = 8$.

In our hybrid design experiments we examined an $S(128, m)$ adder where $m \in \{2, 4, 8, 16, 32\}$. Figures 3.5 and 3.6 show the space utilization and maximum delay of our hybrid designs. The blue and green lines show the performance of each of the classic adder designs, whereas the red and orange lines show the performance of our hybrid design.

### 3.3 Results

Our experiments show that by utilizing a feature-oriented approach, we can rearrange and regenerate the adder characterization to explore the design space that would normally be out of the reach of hardware designers. Using aspects to apply the carry feature made building the hybrid design relatively easy. In total, it only took us a few hours to create our hybrid design. This included designing, coding, and testing. Figure 3.3 shows the breakdown of the code for our hybrid design. This relatively small codebase generates tens of thousands of lines of hardware code for higher bit adders. Totaling only 194 lines, this technique has high code leverage (Figure 3.4) for our endpoints. In the 2 sub-adder case, we generate dozens of lines of code.

Figures 3.5 and 3.6 demonstrate the expanded design space in LUTs and the delay, respectively. Rather than being stuck with the ripple-carry (green) or the carry-lookahead (blue), we provide a space between the two. By having a hybrid design where we can vary the amount of ripple-carry and carry-lookahead, we can better optimize the design for a particular application, rather than just choosing one of the extremes. The results of this study were

<table>
<thead>
<tr>
<th>Type</th>
<th>Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>112</td>
</tr>
<tr>
<td>Aspect</td>
<td>45</td>
</tr>
<tr>
<td>Carry-Lookahead Generator</td>
<td>37</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>194</strong></td>
</tr>
</tbody>
</table>

Figure 3.3: The number of lines of code to generate the hybrid designs.

<table>
<thead>
<tr>
<th>Sub-Adders</th>
<th>Code Leverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>41</td>
</tr>
<tr>
<td>4</td>
<td>17</td>
</tr>
<tr>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>32</td>
<td>8</td>
</tr>
</tbody>
</table>

Figure 3.4: Lines of code generated per generator line of code.
Figure 3.5: The number of LUTs used in the hybrid design.

Figure 3.6: The maximum delay for the hybrid design.
In Chapter 2 we discussed the Chisel elaboration process. Features can be applied at three different phases of the elaboration process: lexical, semantic, and emitting. In optimizing compilers, as the program travels through different passes, the representation of the program becomes more verbose. Thus, it is advantageous to perform a transformation at the first point where the necessary information is available [30]. Similarly, the stage at which the feature best applies is determined by the generator and circuit information needed. To elucidate this, we will make analogies to similar compiler phases.

4.1 Lexical Application

Some features only require *lexical* information to be applied. In a compiler, the parser verifies correct syntax by building a representation of the program as an abstract syntax tree [30]. Some hardware features can be applied simply by modifying the abstract syntax tree of the

Figure 4.1: The levels of design elaboration in Chisel execution.
class Tile(val coreParams: CoreConfig, val nastiParams: NastiBundleParameters, val cacheParams: CacheConfig) extends Module {

  val io = IO(new TileIO(coreParams.xlen, nastiParams))
  val icache = Module(new ICache(nastiParams, coreParams.xlen))
  val dcache = Module(new DCache(nastiParams, coreParams.xlen))
  val core = Module(new Core(coreParams))
  val arb = Module(new MemArbiter(nastiParams))

  io.host <> core.io.host
  core.io.icache <> icache.cpu
  core.io.dcache <> dcache.cpu
  arb.io.icache <> icache.mainMem
  arb.io.dcache <> dcache.mainMem
  io.nasti <> arb.io.nasti
}

Figure 4.2: Code for creating the structure of RISC-V-mini.

hardware generator. Performance counters fall into this category. Regardless of the clock, circuit state, or properties of the elaborated circuit, the implementation of the performance counter remains the same. All they need to know is the label of the signal to count and then where the counter result needs to be accessed.

The hardware generator code contains the necessary information to apply the performance counter features. All hardware is represented as objects within the Chisel language. Creating and manipulating the relationships between these objects can be done at the code level. Adding a new core only requires instantiating a new “core” object. Similarly, a cache could be added to the core through a “cache” object and connecting them in code.

Although any Chisel generator can be manipulated in this way, due to feature entanglement of the decomposition, this is not always straightforward. This necessitates refactoring or building a simpler framework with type-rich hardware constructs. By imbuing the generator with rich type information, we have more information at our disposal to apply features. Take the ALU example from earlier (Figure 2.4). With the current structure of the class, adding support for multiplication or division instructions would require either extremely specific feature application, by hand refactoring, or the implementation of a completely new ALU. With better organization, even as simple as adding methods related to what functionality is being generated, it would be much easier to “hook” into for feature application at the lexical level.
class StructuredAlu(val width: Int) extends Alu {
  val io = IO(new AluIO(width))

  val shamt = io.B(4, 0).asUInt

  val results = ArrayBuffer[(UInt, UInt)]()

  generateAddSub()
  generateShift()
  generateBitwise()

  io.out := MuxLookup(
    io.alu_op,
    io.B,
    results
  )

  def generateAddSub() = {
    results += (ALU_ADD -> (io.A + io.B))
    results += (ALU_SUB -> (io.A - io.B))
  }

  def generateShift() = {
    results += (ALU_SRA -> (io.A.asSInt >> shamt).asUInt)
    results += (ALU_SRL -> (io.A >> shamt))
    results += (ALU_SLL -> (io.A << shamt))
    results += (ALU_SLT -> (io.A.asSInt < io.B.asSInt))
    results += (ALU_SLTU -> (io.A < io.B))
  }

  def generateBitwise() = {
    results += (ALU_AND -> (io.A & io.B))
    results += (ALU_OR -> (io.A | io.B))
    results += (ALU_XOR -> (io.A ^ io.B))
    results += (ALU_COPY_A -> io.A)
  }
}
4.2 Semantic Application

There are circumstances when the code is not sufficient to know how or what hardware must be added to the design. Additionally, we may want to create dynamic structures and features that build up during the elaboration. Rather than specifying these features in code alone, we can represent them as data structures. The lexical level of the generator does not have enough information to apply these features, we need semantic information. It is not enough to know what the syntactic structure of the generator is, we need to know what it means. This is analogous to the intermediate representation of the compiler [30]. The global data structure Chisel produces during elaboration (see Chapter 2), along with program data, can be thought of as a pseudo-intermediate representation of the circuit.

One such example is the hardware finite-state machine (Chapter 6). In order to properly integrate the features into the design, the FSM must evolve have the proper control structures. For a small number of states and transitions, this could be accomplished at the lexical level. However, even at tens, if not hundreds of states, this quickly becomes unwieldy. For Chisel specifically, the syntax for finite-state machines is not easily amenable to feature-orientation. The FSM must also be aware of runtime information about the design. In case of an FSM for a cache, if the design has more than one core, then the FSM may need to generate transitions for cache-coherency. Furthermore, to generate a working FMS, we must have a register that is wide enough to hold the state value, which can only be discovered after by adding features to the FSM. To do this, the specification of the FSM is a data structure, rather than a structural specification in code. The data structure therefore could manipulate the internal representation of the FSM at runtime through provided feature-oriented transformations.

This approach is used frequently in Chisel already, but in a more embryonic form. Designers create hardware objects around the nouns in the decomposition. However, those objects tend to be used to transport ideas around the generator, rather than implementing the verbs or manipulating the hardware. For instance, inside Rocket Chip [9], all the possible instructions that could be generated are packaged into an Instruction object, obscuring which version of the ISA that is being generated. This organization also makes the instructions difficult to extend for future customization of RISC-V. The Instruction does assist the generator in easy transport through the infrastructure. Accessing which instructions to generate follows
```scala
import State._

// The state register
val stateReg = RegInit (green)

// Next state logic
switch ( stateReg ) {
  is ( green) {
    when(io. badEvent ) {
      stateReg := orange
    }
  }
  is ( orange ) {
    when(io. badEvent ) {
      stateReg := red
    }
    . elsewhen (io.clear) {
      stateReg := green
    }
  }
  is (red) {
    when (io.clear) {
      stateReg := green
    }
  }
}

Figure 4.4: A small Chisel FSM example [68].
```
the old if-else methodology. If the instructions were treated as data, the implemented ISA could be more explicit and easier to extend.

4.3 Emitter Application

Some features are best applied after the full circuit has been elaborated when Chisel has emitted the FIRRTL code. Any feature that is better served by knowing the structure of the final hardware falls into the category. During the FIRRTL compilation phase, the restricted FIRRTL has conditional statements and loops removed, resulting in a completely static circuit representation. This space provides opportunities for optimization of the final design.

The adaptive adder in Chapter 3 demonstrates this kind of optimization. The adaptive adder has a larger range of space and power trade-offs than traditional adder designs. Providing the final design with different implementations as features could provide a path to easier fitting to design constraints. Currently, the FIRRTL compiler performs a series of optimizations in the FIRRTL design through several lowering phases [43].

In fact, the original Chisel Aspect library [42] was built to target this exact purpose. The authors proposed injecting design collateral that cross-cut the circuit into the final designs, such as verification libraries, resiliency libraries, and floor planning. Out of all of these examples, floor planning demonstrates the utility of this approach the most. Using this technique, the physical components of the design can be moved around to achieve better power and performance.
Chapter 5

Feature-Oriented Performance Counter Designs

5.1 Introduction

In this chapter, we focus on performance counters and their variations which are defined in the RISC-V architecture specification. However, some applications may not need performance counters, and their omission can result in a smaller footprint and less power. Some applications may need only a subset of the counters, while others may need more counters that are provisioned in the architecture.

Exploring this design space in modern processors, some of which have hundreds of performance counter events [41], by hand is a monumental task. Especially because performance counter events cross-cut the entire design of the chip. Information must not only be collected, but also be routed to the correct location. In this chapter, we present our feature-oriented performance counter system that allows us to incorporate events within the chip without hand-refactoring.

We reformulate performance counters into the following orthogonal features.

Which events are counted? Applications can specify which events in RISC-V are of interest, including architected instructions, microarchitecture events, and cache events. Furthermore, our approach allows the introduction of events and their counters into coprocessors and other logic deployed alongside RISC-V.
When are events counted? As designed, the RISC-V counters always count events when they occur. However, developers are often interested in events only within certain methods or regions of program execution [37]. Our approach allows the introduction of address-range specifications that restrict when events are counted.

5.1.1 Research Questions

To what extent can we modify the generated logic to incorporate new features? Performance counters affect the chip both qualitatively and quantitatively. Qualitatively, to be of any use to programmers, configuration and results of the performance counters must be accessible to the world outside the chip. This can be done either through the ISA with a set of new instructions or through some external configuration outside of the ISA. Quantitatively, the performance counters need to be implemented in general. Signals must be captured, and wires must transmit this information to the appropriate counter. In addition, designers must also choose how many events to support and how many registers to include.

What type information is needed to make the design amenable to feature application? If we are to implement performance counters in an existing characterization of RISC-V, what refactoring need to be done? What types are needed to characterize the performance counters? How can we make the type characterization independent of the hardware characterization?

To what extent can we address crosscutting features in the characterization? What framework do we need to develop to describe the hardware of performance counters in an aspectual way? To what extent can we isolate different features to make the design space more granular? What level of the Chisel toolchain should be targeted for feature application?

5.2 Current Monolithic Design

The current performance counter system in Rocket Chip follows the structure encouraged by traditional hardware definition languages (Section 2.3) rather than the modularity of
an object oriented language like Chisel. Regardless of whether the user actually requires
performance counters, the entire infrastructure to create the system remains in the generator.
This includes all the routing infrastructure needed to get the signals to the counters. In total,
there are 9 separate classes where the performance counter system is implemented, with 3
of those just routing signals. Furthermore, most of the infrastructure is hard coded into
the constructors of the classes, obscuring where the performance counter implementation
actually is.

The undesirable side effects of a monolithic design are exemplified in the performance counter
system\textsuperscript{2}. This monolithic approach makes it not only harder to understand and customize
the performance counter system, but also the whole Rocket Chip system itself.

In this instance, both removing functionality and extending functionality of the performance
counters completely changes the configuration values for the resulting system. This is true
for both individual events and the provided event sets. The bitmask value that configures a
counter to capture an event is determined by the order in which it appears in the generator.
Thus, adding or subtracting events changes the bit patterns for all the other events that
follow it in the code. This can become very complicated if users only want a subset of the
provided events with the bit patterns changing in unexpected ways. Furthermore, generating
hardware based on the execution path creates uncertainty. Generator authors must ensure
that execution occurs \textit{exactly the same way} every time to achieve consistent results.

5.3 Feature Application using Scala Trees

Instead of monolithically including all possible hardware features in a generator, we separate
them out in the generation infrastructure into \textit{feature units} (i.e. aspects) which then can be
\textit{applied} into the generator code base.

In order to automate the process of applying features, we have created a tool in Scala
called \textbf{Faust} (feature application using Scala trees) which can be found at github.com/wustl-
frisc/faust. Faust is a prototype, but provides functionality for capturing features in a custom
DSL and automatic collection of feature dependencies.

\textsuperscript{2}Referenced code can be found at github.com/chipsalliance/rocket-
chip/blob/master/src/main/scala/rocket/RocketCore.scala
5.3.1 Scala Trees

Faust uses Scalamaeta [67] to apply features by manipulating Scala abstract syntax trees (ASTs). Thus, we can directly target the lexical level of the Chisel toolchain. A feature unit may crosscut many different parts of the code base with implementation information that changes many different code modules. For example, a feature that implements an event to be monitored may add IO and connections to many different modules when routing the signal.

To apply features, Faust takes in a directory of Scala code, the feature units themselves, and dependency information. Each file is parsed into its corresponding AST. The AST is then traversed to find a point where feature implementation information needs to be applied, and then the tree is transformed producing a modified AST. This process is repeated until all implementation information is exhausted. In this way, we emulate traditional AOP compilers. Finally, a copy of the original file is saved and a new one with the resulting AST is produced.

Unlike the aspect library in Chisel [42] which operates upon the produced FIRRTL, we are directly manipulating the Scala tree, giving us the ability to influence any part of the generator along with retaining the complete Chisel build system.

5.3.2 Feature DSL

In order to ease the creation of features, we provide a small DSL embedded in Scala. Although Faust is not a proper aspect-oriented compiler [44], we borrow the general syntax found in aspect-oriented programming languages such as AspectJ [29]. The listing 5.1 shows a feature and its syntax.

**Keywords** before (join point) and after (join point) indicate where the advice information will transform the AST. Currently that is either directly before or directly after the join point. extend (class) allows a class to be extended with new type information. insert (code) tells Faust what new implementation information needs to be inserted into the AST. Sometimes a user might wish to refine the join point by using a specific type context in (context) provides this functionality. Finally register adds the advice to the system.
To add a new feature to the system, users only need to extend the Feature class and fill the body with implementation advice. To ensure that all features can be properly implemented, users must also provide dependency information for new features.

5.3.3 Dependency Management

Faust automatically determines and applies the parent features that any child feature depends upon. A JSON file contains a graph of all features and their dependency relations. In a separate JSON file, the end user lists features they would like Faust to apply to the code base.

For each feature in the requested, Faust determines what other features need to be implemented doing a depth-first traversal of the feature graph. Features are added until either the root node has been reached or a feature where all the parents have already been explored.
Thus, we also avoid having to re-traverse portions of the graph already included by other features.

### 5.3.4 Current Limitations

Due to the prototype nature of Faust, it currently has some limitations. First, all of the join points, implementation code, and contexts must be captured as quasiquotes. These are strings that represent ASTs, which makes the implementation of Faust easier. Thus, the features created in Faust are tied to a specific generator implementation (see Chapter 8). Second, new features must be manually added to Faust’s management system. In the future, we would like to have this be an automatic process. Finally, Faust can recognize only pure dependencies. More advanced types of dependencies, discussed in Section 5.4, must be manually handled by the user.

### 5.3.5 Modifications to Rocket Chip

Faust can hook onto individual statements, thus, could apply features inside unmodified Rocket Chip. However, this approach is very fragile, as any change to a statement would break the feature application. To facilitate better application of features, we have made several changes to Rocket Chip that make the generator designs more modular.

The current design of Rocket Chip treats `Module` classes as direct analogs for Verilog modules. Instead, we propose thinking of classes as hardware types that perform various generation tasks. A class that implements a hardware type should only extend the `Module` class when appropriate. Listing 5.2 shows this technique in the context of the CSR structure and how we have utilized it in the implementation of our Performance Counter system. This pattern is revisited later in Chapter 7.

In the version of Rocket Chip we modified, the `CSRFile` class in Rocket Chip contains 907 lines of code. By introducing this structure, generator users do not need to sift through all the code to find exactly where new features can be inserted, giving us a robust set of join points in which to easily integrate features into the system.
Using Faust, we have completely refactored the Rocket Chip performance counter system to be feature-oriented. In addition, we have also enhanced the functionality of the current system and provided features that extend or modify the system that chip designers might use.

Our *Base System* is the modified version of Rocket Chip discussed in Section 5.3. The base system does not contain any generation infrastructure for the performance counters.
5.4.1 Feature Decomposition

We are careful to manage the dependencies between the features in our system. In addition to the Base System itself, none of the other features can produce a functional system on their own. Figure 5.3 maps out all the current features in our system and their dependencies. The majority of the current dependencies are pure dependencies. However, two of our features are mutually exclusive and cannot exist in the system at the same time. Right now, mutually exclusive features must be handled manually.

We borrowed our feature schema from prior work on feature-oriented design by Hunleth and Cytron [40]. Following their schema, we have two types of features in our system. Abstract
Features are features that provide infrastructure, but do not complete the primary functionality on their own. Abstract Features must always be augmented by a Concrete Feature. A Concrete Feature provides primary functionality since its dependencies are satisfied.

Counters

Counter System  The Counter System feature is the only abstract feature that exists in the system. This provides the necessary infrastructure into Rocket Chip to realize the performance counter system, but leaves out creation of the performance counter registers.

Direct Counters  The Direct Counters build the standard performance counters that are found in Rocket Chip. End users can still choose how many performance counters they wish to include in the system.

Address Restricted Counters  Instead of collecting event information over the whole address range, the Address Restricted Counters produce counters that are inhibited unless the program counter is within a certain address range. This feature allows users to customize the address range. Furthermore, this is managed completely outside of the architecture. The architected interface does not change.

Events

Instruction, Microarchitectural, & Cache Events  These three features reimplement the standard event sets from Rocket Chip. However, unlike the original events and event sets, the order in which they appear in the code does not change the bit values to configure the events. Now, each event and event must be configured with a bit value to distinguish itself. The advantage here is that the events and event sets can be applied to the codebase in any order and events can be added or subtracted without affecting any other events. In fact, the way in which we have split events into these sets is completely arbitrary and could even be atomized further.
In order to ensure that correct interfaces have been created, we provide correctness checking at generation time. If one or more events share the same bit mask, then the generator will throw an exception, telling the end user that they have created an invalid configuration.

**Accumulator Event**  We recognize that this feature is very limited, but we include it to demonstrate extra functionality that can quickly be included via our feature-oriented approach. The Accumulator Event feature routes a signal out of an RoCC \[13\] accumulator accelerator (included in Rocket Chip) and into the core for performance counting. This feature provides a template for others to include performance counter information from their accelerators as well.

### 5.5 Results

Here we characterize the behavior of design endpoints in terms of area utilization. Figure 5.4 shows 11 separate endpoints generated using Faust as well as comparisons to Rocket Chip. All LUT results are normalized to the endpoint “No System”, which is our system without any counters applied. Our simulation and testing was completed using Chipyard \[4\] and our system is realized within a design based on the TinyRocketChip provided by Chipyard. We targeted the XC7A35T-1CPG236C FPGA which is in the Atrix-7 family of FPGAs from Xilinx. All implementation was done using Vivado 2019.2. The results of this study were published at the Fifth Workshop on Computer Architecture Research with RISC-V (CARRV) in June 2021.

#### 5.5.1 Rocket Chip vs. Our System

The resulting zero counters systems are within 0.1% of each other in LUT usage. Furthermore, when recreating the full performance counter system from Rocket Chip we find similar results. These two systems are within 0.3% of the same area usage. Thus, we find that feature-orienting the performance counter system does not add any space penalty. While the feature set we are examining here is small, we are still effecting a large portion


<table>
<thead>
<tr>
<th>Counter Type</th>
<th>Events</th>
<th>LUTs</th>
<th>Normalized LUTs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct</td>
<td>Inst Events</td>
<td>26253</td>
<td>1.09</td>
</tr>
<tr>
<td>Direct</td>
<td>Inst Events Micro Events</td>
<td>26521</td>
<td>1.10</td>
</tr>
<tr>
<td>Direct</td>
<td>Inst Events Micro Events System Events</td>
<td>26636</td>
<td>1.11</td>
</tr>
<tr>
<td>Direct</td>
<td>Inst Events</td>
<td>26593</td>
<td>1.11</td>
</tr>
<tr>
<td>Direct</td>
<td>Accelerator</td>
<td>26521</td>
<td>1.10</td>
</tr>
<tr>
<td>Direct</td>
<td>Inst Events Micro Events System Events</td>
<td>26636</td>
<td>1.11</td>
</tr>
<tr>
<td>Direct</td>
<td>Inst Events</td>
<td>26636</td>
<td>1.11</td>
</tr>
<tr>
<td>Direct</td>
<td>Accelerator</td>
<td>26521</td>
<td>1.10</td>
</tr>
<tr>
<td>PC</td>
<td>Inst Events Micro Events System Events</td>
<td>26555</td>
<td>1.11</td>
</tr>
<tr>
<td>PC</td>
<td>Inst Events Micro Events System Events Accelerator</td>
<td>26872</td>
<td>1.12</td>
</tr>
<tr>
<td>PC</td>
<td>Inst Events Accelerator</td>
<td>26365</td>
<td>1.10</td>
</tr>
<tr>
<td>PC</td>
<td>Inst Events</td>
<td>26378</td>
<td>1.10</td>
</tr>
<tr>
<td>No system</td>
<td></td>
<td>24025</td>
<td>1.00</td>
</tr>
<tr>
<td>RocketChip (0 Counters)</td>
<td></td>
<td>24056</td>
<td>1.00</td>
</tr>
<tr>
<td>RocketChip (All Counters)</td>
<td></td>
<td>26542</td>
<td>1.10</td>
</tr>
</tbody>
</table>

Figure 5.4: Space of our endpoints in LUTs.

of the design with our features. This result demonstrates efficacy for further applications of feature-oriented design for hardware.

### 5.5.2 Design Endpoint Variations

Although our system can produce numerous design endpoints, in Figure 5.4 we show the interaction of features through different design endpoints. Clearly, the infrastructure of the performance counter system takes the most space, rather than the events themselves. Eliminating or restricting this could result in space savings, especially for embedded systems.

**Instruction Events vs. All Events** Regardless of the type of counters being used, there is about a 2% increase in area between just the *Instruction Events* and *All Events* endpoints, showing a clear advantage for letting end users choose what features should be included in the performance counter system. Easy savings in area are not realized when all features are included by default. By taking this additive approach, Rocket Chip could not only provide more flexibility in the design but could automatically save designers space that would take manual effort to do currently.

**Direct vs. Address Restricted Counters** Regardless of the type of events in the system, the *Address Restricted Counters* feature only uses 1% more space than the *Direct Counters*. A feature like this is extremely useful for programmers as many times they only
wish to performance monitor sections of their code. By completing this action exclusively in hardware, programmers do not have to perturb the system, leading to more accurate results. Our feature-oriented approach allows Faust to insert this using a single line of code which creates a new class that is only 7 lines. By having an agreed upon way to add features to the system, we help guide the creation of new features to affect the rest of the system as little as possible.

**Comparing Features** A prime advantage of this approach is shown by this analysis. Since *elided features do not exist in the system*, when we add a feature we can be more certain about what how each feature effects the system. Furthermore, paring this approach with Faust means that we can quickly compare different design endpoints with accuracy.
Chapter 6

Feature-Oriented Construction of Finite State Machines

6.1 Introduction

Finite-state machines are ubiquitous in computing. Ranging from software designs for image cryptography [3], modeling cache security states to find side channel attacks [83] and modeling cache coherence protocols [54], to hardware components such as predictors [65], packet processing [61, 33], and creating cache coherence protocols [58].

In the case of [58] for producing cache coherence protocols, multiple and hierarchical cache designs can end up with hundreds of cache states. Producing these FSM with hundreds of states and transitions creates the potential for introducing many bugs. Instead, a feature-oriented generative design based on simple specifications could produce these designs.

While Chisel does contain syntax for creating finite state machines, it is not amenable to feature-orientation. Consider adding a new state to the Chisel example in Figure 4.4. This syntax does not have clear hooks to grab onto for adding implementation information. In addition, adding a new state may require that the register holding the current state become wider. Instead of explicitly defining the finite state machine, we chose to dynamically generate it based on advice. This not only solves the stated problem, but also decouples the characterization of the FSM from the generated hardware. This results in a powerful way to generate very large FSMs.
6.1.1 Research Questions

To what extent can the control structures be modified to incorporate new features? As described earlier, finite state machines can appear to be quite rigid. How can we apply transformations to finite state machines that maintain their properties? What generation leverage can we get out of transformations for applying features? How can we describe both qualitative and quantitative features of a finite state machine?

What types are necessary to make control structures amenable to feature application? What “hooks” are necessary to add to finite state machines for feature application? What type information must be added to assist with transformations? How should finite state machines be represented within a feature-oriented framework?

How can cross-cutting concerns be addressed in the characterization of control structures? What framework do we need to develop to describe finite-state machines in an aspectual way? To what extent can we isolate different features to make the design space more granular? What level of the Chisel toolchain should be targeted for feature application?

6.2 Generative FSM specifications

We next illustrate our two FOP constructs for generating complex FSMs. The first uses aspect-oriented advice to incorporate features selectively into an FSM. The second builds a cross-product FSM from the synchronous simulation of smaller FSMs.

6.2.1 Feature introduction via AOP

As an example of a featureful hardware design, we consider an FSM implementation of a vending machine. A state in our design carries the necessary (Scala) traits to represent its role in the machine’s operation: the funds inserted and the potential products dispensed. Our generative approach described below offers the following advantages over a monolithic design:
• The design itself is simpler and clearer when described using FOP. A monolithic design that tangles all features can be realized, but the resulting FSM does not readily make the features apparent. Moreover, the work to create that monolithic implementation is tedious and error-prone.

• Modification of the FSM is greatly simplified. For example, introduction of a new value of coinage automatically creates the necessary additional states and transitions.

• Scala traits allow elegant expression of an application’s behaviors in support of FOP hardware design.

For this example, the features of interest for a vending machine are as follows:

Add Currency introduces a value of coinage.

Dispense Product introduces a vendible item and its price.

Print Funds causes the machine to display the total funds after each state change.

Insufficient Funds introduces a prompt to advise the consumer to insert more funds to buy a particular item.

Change Return introduces a button (input to the FSM) that causes the machine to return unspent funds.

Peanut Warning requests confirmation of purchase for items that contain peanuts.

Buy More allows the consumer to continue purchasing items if funds remain in the machine. The Change Return feature, if present, allows the consumer to request return of the remaining funds.

The dependencies of these features are shown in Figure 6.2, but this graph is not needed for construction: the advice for a given feature is applicable only when its associated join points exist in the FSM. As is typical with aspect-oriented approaches, all advice is presented to a weaver (our runtime library for Chisel), and the aspects are continually applied until no changes occur.

For example, the advice for Add Currency of coinage \( k \) specifies that for any state representing that \( n \) cents have been inserted, a state representing \( n + k \) cents must exist, with a
transition from state $n$ to state $n + k$ based on the insertion of coinage $k$. This advice fails to terminate if not capped by some upper bound on funds, which could be related to the most expensive product sold. For example, Figure 6.1 shows a machine that

- Accepts only 5 cent coinage
- Accepts up to 15 cents
- Vends peanuts that cost 10 cents

The FSM is automatically generated by the advice to Add Currency and Dispense Product. Continuing with this example, consider the Buy More feature, intended to incentivize consumers to spend more money. This feature causes the machine to retain remaining funds after a purchase to encourage subsequent purchases. Without this feature, the machine in Figure 6.1 would return 5 cents if 15 cents are used to purchase peanuts costing 10 cents. With the feature, the 5 cents of remaining value would be held by the machine for subsequent purchases. The advice for this feature modifies every purchase to move to a state representing currently held funds. In Section 6.3.1 we discuss application of other features to this FSM.

A monolithic approach requires designers to specify all states and transitions for each feature subset, which is tedious and error-prone. With our approach, designers can verify the correctness of much smaller designs and then obtain much larger generative designs that are correct by their construction.

In terms of leverage, consider an FSM for which there are $n$ orthogonal and independent features. A valid system could thus be written or generated with or without each of those $n$ features. This leads to $2^n$ feature-specific implementations. While it is unlikely that each of those implementations would find an application, the ability to generate any of them automatically offers significant leverage.

6.2.2 Generating FSMs via cross-product composition

Nim [80] is a broad class of impartial mathematical strategy games, which traditionally involve multiple heaps of tokens (e.g., sticks) and two or more alternating players. The
Figure 6.1: FSM for a vending machine that accepts 5 cent coins and dispenses peanuts that cost 10 cents.

Figure 6.2: Dependencies between vending machine features.
current player removes an allowable number of sticks from a subset of the heaps. The winner is usually defined as the player taking the last token. In a *misère* version of the game, that player would lose. While we initially approached this game with the ideas in Section 6.2.1, states in this game represent the number of tokens present in each heap and the current player’s turn. Aspectual features should be additive, expanding the behavior of their targets, and while cross-cutting, they do not typically completely rewrite their targets. With Nim, the machine is changed throughout by the addition of an additional player or heap.

However, we realized that construction of even the most basic game of Nim can be regarded as the composition or simultaneous operation of two simpler machines: one that represents only the allowable subtractions of tokens in a heap (such as the 5-token heap shown in Figure 6.4) and one that represents only the alternation of players (such as the two-player alternation shown in Figure 6.5). Transitions not shown in those machines are errors, such as Player A taking two consecutive turns. By a composition technique we present in Section 6.4.1, we obtain the basic game of Nim shown in Figure 6.6. That algorithm simulates the simultaneous, lock-step execution of the machines shown in Figures 6.4 and 6.5.

Following is our feature decomposition of Nim:

**Heap Bounds** encodes the initial and winning number of tokens for each heap.

**Legal Moves** encodes permissible combinations of adding or removing tokens from each heap.

**Number of Players** specifies how many players take turns in the game.

**Win Type** specifies whether the game is *misère* play or normal play.

The dependencies of these features is shown in Figure 6.3.

The game is won when both conditions are met:

- Players alternate correctly, as in Figure 6.5. For example, the sequence ABA leads to an accept, but the sequence AAB cannot.

- All tokens have been taken, for example, using the sequence 212 in Figure 6.4.
Formally, in the theory of regular languages, acceptable inputs for our problem are in the *concatenation* of the languages of these two machines. Given the sequence ABA and 212, each accepted by its respective machine, the concatenation ABA212 is a string in the concatenation of the two languages. While this correctly represents the sequence of inputs that causes somebody to win, the timing of actions associated with transitions in the two machines does not coincide properly. For example, it is not possible in the concatenation to determine *who* won the game when the second machine moves to its accepting states, where zero tokens remain.

We prefer to process the inputs more naturally as (A,2), (B,1), and (A,2), so that the player making the move and the number of subtracted tokens are processed in lock-step. In this way, any actions associated with each move will be properly taken upon that move. Similarly, if the game had 3 heaps in play instead of a single heap, we want all heaps to be appropriately modified by each turn. Concatenation would focus on one heap until it is exhausted before moving to the next heap. Finally, consider that the first machine accepts ABAB which would be inappropriate if the second machine accepts 212, since only 3 moves are made. The lock-step execution we require is over *same-length* inputs for each machine.

In support of the semantic actions occurring as they should, and requiring inputs of the same length, we present in Section 6.4.1 an algorithm for constructing a particular *cross product* of two FSMs. For our example here, the resulting FSM is shown in Figure 6.6, which makes moves with the proper timing and appropriately distinguishes a win by A from a win by B.

In terms of leverage, consider the cross-product generation of an FSM from two identical FSMs each of size $m$ (states+transitions). For example, we generate a 2-heap version of Nim by taking the cross product of two such FSMs, as described in Section 6.4.1. The resulting machine is of size $O(m^2)$. An $n$-way cross product generates a machine of size $O(m^n)$, where
Figure 6.4: FSM for a 5-token heap that allows one or two tokens to be removed in a turn.
$m$ is viewed as a constant here. The structures we can generate with this approach are (in the limit) exponential in the size of their specifications.

### 6.3 Hight-Level Overview

An FSM $M$ is typically defined as follows:

$$M = (Q, \Sigma, \delta, q_0, F)$$

where $Q$ is a set of states, $\Sigma$ is a set of tokens, $\delta$ is the transition function, $q_0$ is the start state, and $F$ is the set of accepting states. The symbol $\lambda$ denotes the empty string. When an FSM is drawn as a graph as in Figure 6.8, states are shown as nodes and transitions as labeled, directed edges. The start state receives an edge with no source, and an accepting state is drawn with two concentric circles.
Figure 6.6: The resulting Nim finite-state machine. The edge transitions are labeled with the player who acts to take the specified number of tokens. The state is labeled with the player who just completed a turn and the number of remaining tokens.
Decidability and Testing

For regular languages and (thus) FSMs, questions of interest in machine behavior are decidable [71]. For example, consider a feature that is intended to extend the behavior of a design, so that all previous inputs are allowed, but the feature introduces some other allowable inputs. An example of this is an FSM that scans characters to parse them as decimal numerals. A feature might extend that syntax to accommodate hexadecimal numerals for inputs that begin with 0x. With our approach, the introduced feature creates a new FSM, whose language is provably a proper superset of the original FSM’s language. The ability to reason about modified FSMs and their behavior can greatly reduce the need for testing. We do not rely on this advantage in our work, but this is one reason FSMs are attractive for hardware designs.

6.3.1 Cross-cutting features

We follow [64] in the treatment of aspects for FSMs. Essentially, a state is like a method and a transition between states is like a method call. The usual forms of before, after, and around advice are available (cf. AspectJ [29]) A cross-cutting feature is implemented using advice that modifies an FSM’s behavior before, after, or during a transition between states.

As described below, a feature consists of advice applied to pointcuts of an FSM, which can formally change the language of the machine. More broadly and usefully, the advice can affect actions taken by the machine as its inputs are processed.

As an example to show the advantages of an AOP approach for FSMs, consider the inclusion of a write-back feature for a cache. The feature requires modification of the cache design in multiple places:

- Each cache line must contain a dirty bit.
- A line’s dirty bit must be set if the program modifies any byte in that line.
- When a line is replaced, the dirty bit must be consulted to determine if the line needs to be written to the backing storage.
Without an AOP approach, the FSM is modified by hand in various portions of its specification to realize the above behaviors. If subsequently a write-through approach is desired, the write-back logic must be edited out and the write-through logic deployed. This is an error-prone and tedious activity. With an AOP approach, the designer can easily change from write-back to write-through by simply selecting the desired feature for inclusion. For write-back, the above FSM modifications are expressed together in a single feature, whose inclusion affects the FSM appropriately as described above. The elements of the feature are:

- A pointcut specifies where in the FSM changes should occur. The pointcut then yields a set of states.
- join points, which are specific states or transitions in the FSM, at which modifications to the FSM occur.
- Advice in the form of FSM modifications is applied.

**Pointcuts** Pointcuts denote a set of states or a set of transitions in an FSM, typically conditioned on some predicate $p$. Evaluated at run-time when a design is generated, $p$ can be any Boolean-valued function that selects items of interest, typically based on types, traits, or values of instance variables in the hardware-generating program. When a feature is newly conceived, it is likely that the base implementation and extant features will be refactored to articulate and expose necessary types (hooks) for the new feature’s inclusion. However, the effort is worthwhile because the refactoring raises the abstraction level of the specification and allows for richer and more precise pointcuts.

State pointcuts are analogous to a set of function bodies in a program. Just as a function body is unique within a program, each state join point within the pointcut will also be unique. Pointcuts that specify transitions on tokens are analogous to a set of function calls. Unlike function bodies, calls to a given function may appear multiple times within a program. This is also true for a token within an FSM. Thus, the resulting pointcut will contain all the transitions where the predicate $p$ was satisfied.

For example, the **Print Funds** feature can be generated in a vending-machine FSM through after advice applied to any token that adds value to the machine. Such properties are supported nicely in Scala using traits. To implement this feature, the base code likely
Figure 6.7: The resulting FSM of the application of Print Funds to the FSM from Figure 6.1.

requires refactoring to include the value trait. In AOP terminology, a pointcut yields a set of join points at which advice is applied. A join point associated with the above example would be a single token “5”, such as the one between state “10” and “15”, at which the value increases in the machine by “5” cents. Because this is an after pointcut, the join point has context that includes the state “15” that follows the token “5”, as shown in Figure 6.1.

Advice Continuing the analogy, an FSM is essentially a set of function bodies (states) where function calls (transitions) occur at the end of each body. For our purposes, advice is inserted into the execution flow of an FSM to implement a given feature. The advice affects the actions performed before, during, or after a state, including modification of transitions to target new or other states.

We consider the usual forms of around, before, and after advice (cf. AspectJ [29]). Around advice takes the join point and replaces it with something else. States replace states, and transitions replace transitions. Before and after advice come in the form of token-state or state-token pairs, in order to add a new path through the FSM. Figure 6.8 shows the effect of different types of advice on a simple FSM. Before advice (q3, e) on q1 takes all paths into q1 and directs them into q3, on e the machine transitions to q1. The reverse is true for after advice (e, q3) on q1. All paths leaving q1 now leave q3 and the transition from q1 to q3 takes place on e. Similarly, before advice (e, q3) on a places e going into q3 and an out of q3. The reverse is true for after advice (q3, e) on a. In each of these cases, e can be the empty string λ if the behavior of the FSM should change but not require additional input to do so.
Continuing with our example, in the Print Funds feature, a new state is inserted following each token “5” in the FSM. The exact print statement generated by the advice is determined by the context contained within the join point. Print \( \texttt{c15} \) is generated because the state “15” follows the token “5” discussed earlier. Furthermore, to prevent infinite application of features, the advice for Print Funds checks to see if the context in the join point is already a printing state. If so, no advice is applied. Like pointcuts, advice is written in Chisel/Scala.

Not only can advice insert new states, but it can also insert new symbols as well. Consider the Peanut Warning feature as an example. The pointcut is predicated upon a dispense state having a “peanut” trait. Because this is a before pointcut, each join point has context that includes transition information that targets the state. In Figure 6.9 this is \( 10 \xrightarrow{\text{Peanut}} \) and \( 15 \xrightarrow{\text{Peanut}} \). The advice will insert a new “Contains Nuts!” state and “Accept” token for each of the join points. It also inserts a new transition on the “Reject” token whose destination is determined by the context contained in the join point.
Figure 6.9: The resulting FSM of the application of Peanut Warning to the FSM from Figure 6.1.

Context Aware Advice

When advice is applied, there is information associated with the join point that provides the immediate context around it. State advice is aware of the paths going into and out of the state it is affecting, and token advice is aware of the states preceding and seceding the transitions each token is involved in. For example, the before advice on state q1 in Figure 6.8 could have two different paths placed before q1 depending on if it was affecting a or c. This allows for dynamic and context-aware advice creation for changing the control structures for hardware.

6.4 Aspect-Oriented Finite-State Machine Library

We have incorporated all of the above mechanics into a software library. In this instance, features were applied at the semantic level. Because we targeted Chisel for hardware generation, this library is also built in Scala. As is, it can be dropped into any Chisel hardware generator to construct and generate hardware FSMs. The software library contains a set of base classes, FSM, state, and token that can be arbitrarily extended by hardware designers to suit their applications.

Furthermore, the library fully handles all feature application through our Weaver class (in AOP, code is “woven” into a codebase). Advice may introduce new join points within the FSM, making it necessary to reapply the aspects that contain those join points in their
class PrintFunds extends Aspect[NFA] {
  def apply(nfa: NFA) = {
    val tokenPointcut = Pointcutter[Token, Coin](nfa.alphabet, token =>
      token match {
        case t: Coin => true
        case _ => false
      })

    AfterToken[Coin](tokenPointcut, nfa)((thisJoinpoint: TokenJoinpoint[Coin], thisNFA: NFA) => {
      var value = thisJoinpoint.out.asInstanceOf[ValueState].value
      thisJoinpoint.out match {
        case s: PrinterState if (s.action == "Funds:" + value.toString) =>
          (None, thisNFA)
        case _ => (Some((PrinterState("Funds:" + value, value, false), Lambda)), thisNFA)
      }
    })
  }
}

Figure 6.10: The implementation of the Print Funds feature in Foam.

pointcuts. As is the case for other aspect compilers, our weaver continues to apply advice until the resulting FSM is the same as the previous iteration. It is thus possible to write an aspect that applies to an FSM without end, but standard advice-authoring care will prevent that. Figure 6.10 shows the implementation of the Print Funds feature in Foam for the FSM vending machine.

### 6.4.1 FSM feature composition using a cross product

Figure 6.11 we describe a generative cross-product technique (due to Harel [34]) algorithm for producing the lock-step, cross-product FSM from two deterministic FSMs. The resulting machine’s states are combined by taking a state from each of the two input FSMs. We call such a state a combined state, and it represents arrival at each of its constituent states in each of the input FSMs. Highlights of the algorithm are as follows:

- When L is a queue, the algorithm performs a simultaneous breadth-first search of the input FSMs, using L to hold combined states that still need to be processed.
Inputs: $A = (Q_A, \Sigma_A, \delta_A, q_A, F_A), B = (Q_B, \Sigma_B, \delta_B, q_B, F_B)$

Output: $C = (Q_C, \Sigma_C, \delta_C, q_C, F_C)$

1: $q_C \leftarrow (q_A, q_B)$
2: $Q_C \leftarrow \{q_C\}$
3: $L \leftarrow \{q_C\}$  \{Work queue\}
4: $P \leftarrow \emptyset$  \{Already-processed states\}
5: while $L \neq \emptyset$ do
6:    $s = (s_A, s_B) \leftarrow$ dequeue($L$)
7:    $P \leftarrow P \cup \{s\}$
8:    for $(s_A \xrightarrow{a} d_A \in \delta_A) \times (s_B \xrightarrow{b} d_B \in \delta_B)$ do
9:        $d \leftarrow (d_A, d_B)$
10:       if $d \notin P$ then
11:           $L \leftarrow L \cup \{d\}$
12:       end if
13:       $\delta_C(s, (a, b)) \leftarrow d$
14:       if $d_A \in F_A \land d_B \in F_B$ then
15:          $F_C \leftarrow F_C \cup \{d\}$
16:       end if
17:    end for
18: end while

Figure 6.11: Construction of the Cross Product FSM. The inputs $A$ and $B$ must be deterministic; the output $C$ is deterministic.

- The algorithm is correct no matter the order in which unprocessed combined states are considered. Thus, $L$ could be a set rather than a queue and the algorithm is still correct.
- $L$ is initialized with the combined state of the input machines’ starting states.
- When a combined state $(s_A, s_B)$ is taken from $L$, the transition functions of the input machines are consulted to find a possible successor state $d$.
- Combined state $d$ is then added to the work queue if it has not yet been processed.
- The set $P$ keeps track of combined states that have been processed.
- From the above, it is clear that a combined state is placed on $L$ at most once.
- At line 13 of Figure 6.11, the algorithm is modifying $\delta_C$ to add the transition $s \xrightarrow{(a,b)} d$. This modification must make concomitant changes to $Q_C$ and $\Sigma_C$, potentially introducing the new state $d$ and new symbol $(a, b)$, respectively.

55
The combined state \( d \) is marked as an accepting state in FSM \( C \) if both of its components are accepting states.

Consider this algorithm applied to Figures 6.4 and 6.5, with \( L = \{ (\text{Player A}, [5]) \} \).

- Currently \( s = (\text{Player A}, [5]) \)
- \( s \) is added to \( P \) so that it will not be considered again for processing.
- We consider the Cartesian product of transitions from state Player A in the player machine and from state [5] in the heap machine.
- This yields
  \[
  \text{Player A} \xrightarrow{B} \text{Player B} \times [5] \xrightarrow{\text{Take 1}} [4]
  \]
  and
  \[
  \text{Player A} \xrightarrow{B} \text{Player B} \times [5] \xrightarrow{\text{Take 2}} [3]
  \]
- Consider \( (\text{Player A}, B, [5], \text{Take 1}) \) \( \rightarrow \) (Player B, [4])
- The combined destination state is \( (\text{Player B}, [4]) \not\in P \).
- It is therefore added to the work queue \( L \).
- \( (\text{Player A}, [5]) \xrightarrow{(B, \text{Take 1})} (\text{Player B}, [4]) \) is added to the cross-product machine’s transition function.
- State \( (\text{Player B}, [4]) \) is added to \( Q_C \).
- Token \( (B, \text{Take 1}) \) is added to \( \Sigma_C \).
- Because [4] is not an accepting state, \( (\text{Player B}, [4]) \) is not added to \( F_C \).

The next transition \( (\text{Player A}, B, [5], \text{Take 2}) \rightarrow (\text{Player B}, [3]) \) is processed similarly:

- The combined destination state is \( (\text{Player B}, [3]) \).
- Because the combined destination state is not in \( P \), it has not been processed yet, so it is added to \( L \).
• Transition \((\text{Player } A, [5]) \xrightarrow{(B, \text{Take } 2)} (\text{Player } B, [3])\) is added to the cross-product machine’s transition function.

• State \((\text{Player } B, [3])\) is added to \(Q_C\).

• Token \((B, \text{Take } 2)\) is added \(\Sigma_C\).

• Because [3] is not an accepting state, \((\text{Player } B, [3])\) is not added to \(F_C\).

### 6.5 Case studies and results

Here we present three case studies to demonstrate the generative ability of our framework: a Vending Machine, and SIMD cache coherence.

#### 6.5.1 Vending Machine

We implemented all the features from Section 6.2.1 in our library. The resulting FSMs were then emitted as Verilog. The Verilog was synthesized on a xc7a35tcpg236-1 FPGA using Vivado 2022.1. Below we report the number of generated states, transitions in the FSM, and the space in LUTs that the FSM took up on the FPGA.

Figure 6.12 shows the results for different endpoints generated by our library. For our tests, we held the currency threshold at 100. Every machine contains 5, 10, and 25 value coins; and 4 products of value 25, 50, 75, and 100. This is captured by None. In the first set of results, each feature is shown by itself. Even single features can greatly increase the components in the FSM. The Buy More feature (denoted B) by itself more than doubles the number of states and transitions. This impressive leverage is further exemplified when combining features.

In these cases the number of states increase by 2.5x in the simplest endpoint up to 4.6x in the most complex, and the transitions by 2.5x and 5x respectively. Recall, this is in a relatively simple vending machine that can only accept up to 100 units of value. Simply doubling the amount of accepted value to 200 creates a machine with 284 states (10.5x increase over the base) and, 3113 transitions (16x increase over the base). However, this is accomplished in
Features | States | Transitions | LUTs  
-------|--------|-------------|------
None   | 27     | 208         | 24   
P     | 47     | 368         | 38   
I      | 47     | 368         | 46   
C      | 48     | 423         | 70   
W      | 33     | 320         | 60   
B      | 57     | 448         | 60   
PI     | 67     | 528         | 73   
PICB   | 118    | 1053        | 186  
PICW   | 94     | 1023        | 160  
PICWB  | 124    | 1353        | 220  

Figure 6.12: Number of generated states, transitions, and LUTs depending on selected features. The features are as follows: P = Print Funds, I = Insufficient Funds, C = Change Return, W = Peanut Warning, and B = Buy More.

our library with relatively few lines of code. The largest feature in terms of code is Peanut Warning, which is implemented in just 39 lines.

Despite the growing number of generated states and transitions as the features increase in Figure 6.12, the resulting hardware resources, in this case Look Up Tables (LUTs), used by the FSM are relatively modest. This is because hardware synthesis tools can represent states using a linear encoding scheme. For an FSM with \( n \) states, each state takes only \( O(\log n) \) space when encoded as an integer. However, the specification of that circuit to a synthesis tool must be expressed state-by-state. If the number of transitions per state is bounded by a constant, then the specification (e.g., lines of Verilog) takes \( O(n) \) space. Our generative approach to hardware FSMs opens hardware designers to implementing much larger FSMs than are currently sustainable with a hand-coded approach.

### 6.5.2 Nim

We implemented all the features described in Section 6.2.2 and studied three variations of Nim: traditional Nim, the subtraction game, and circle Nim. All results assume each game is misère play with at least one heap and one player. We further assume that the players alternate in a round-robin fashion. The details of each variation are described below, along with the number of generated states and transitions within the created FSMs.
Figure 6.13: Number of generated states and transitions by selected features for traditional Nim.

**Traditional Nim** The classical game of Nim contains two players and three heaps, though there are no restrictions on the number of players or quantity of heaps. The defining characteristic of traditional Nim is that, each turn, the current player may only remove sticks from a single heap of their choosing, and must take between 1 and all the remaining sticks within that heap. The game ends when all heaps contain zero sticks.

Figure 6.13 shows the results for different endpoints of the traditional game of Nim, varying both the quantity of heaps, the number of sticks within each heap, and the number of players. In the simplest variation, containing only a single heap with three sticks and one player, a total of six states and 30 transitions were generated. Juxtapose this with the most complex variation, containing three heaps – with three, four, and five sticks, respectively – and four players: this variation contains 419 states and 20950 transitions. With minimal changes to the specification of the game, this represents a nearly 70x increase in the number of states and 698x increase in the number of transitions. For a fixed number of players, the number of transitions experiences growth by an order of magnitude, moving from 3 to 4 to 5 heaps.

**The Subtraction Game** This game is a single-heap variation of Nim, usually played by a small group of players. In our analysis, we fix the starting number of sticks to 21 each game. Each turn, the current player removes between one and a set number of sticks from the heap, with the game ending when the final stick is removed.
<table>
<thead>
<tr>
<th>Allowed Moves</th>
<th>Players</th>
<th>States</th>
<th>Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1 ✓ -2 ✓ -3</td>
<td>1</td>
<td>24</td>
<td>72</td>
</tr>
<tr>
<td>✓</td>
<td>2</td>
<td>24</td>
<td>96</td>
</tr>
<tr>
<td>✓</td>
<td>4</td>
<td>24</td>
<td>144</td>
</tr>
<tr>
<td>✓ ✓</td>
<td>1</td>
<td>24</td>
<td>96</td>
</tr>
<tr>
<td>✓ ✓</td>
<td>2</td>
<td>45</td>
<td>270</td>
</tr>
<tr>
<td>✓ ✓</td>
<td>4</td>
<td>81</td>
<td>810</td>
</tr>
<tr>
<td>✓ ✓ ✓</td>
<td>1</td>
<td>24</td>
<td>120</td>
</tr>
<tr>
<td>✓ ✓ ✓</td>
<td>2</td>
<td>45</td>
<td>360</td>
</tr>
<tr>
<td>✓ ✓ ✓</td>
<td>4</td>
<td>83</td>
<td>1162</td>
</tr>
</tbody>
</table>

Figure 6.14: Number of generated states and transitions by selected features for the Subtraction game. Every game is played with a single starting heap of 21 sticks.

Figure 6.14 shows the results for different endpoints of the subtraction game, varying the number of sticks permissible to take each turn and the number of players. In the simplest variation, with there only being one player allowed to take one stick each turn, the resulting FSM contains 24 states and 72 transitions. By comparison, the most complex variation considered has four players and allows the player to take 1, 2, or 3 sticks each turn, with the corresponding FSM having 83 states and 1162 transitions. This marks an almost 3.5x increase in the number of states and 16x increase in the number of transitions. Empirically, the primary driver for complexity in these variations is increasing the number of heaps, which the subtraction game lacks; however, the ability to generate such a breadth of variations for a single game type with minimal code changes is a huge lever.

**Circle Nim** Circle Nim is given its name due to the layout of its heaps: the heaps are placed around a circle, with their adjacency affecting gameplay. It is traditionally played with a finite number of heaps, each containing a single stick, and two players. Each turn, the current player is allowed to take the sticks from between one and a pre-set number of consecutive heaps. We allow the player to take from between one and three consecutive heaps.

Figure 6.15 shows the results for different endpoints of circle Nim, varying both the number of players and the number of heaps. In the simplest variation, with three heaps and a single player, 10 states and 110 transitions are generated. In the most complex variation, with
<table>
<thead>
<tr>
<th>Heaps</th>
<th>Players</th>
<th>States</th>
<th>Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>10</td>
<td>110</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>15</td>
<td>255</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>17</td>
<td>340</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>66</td>
<td>1320</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>113</td>
<td>4294</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>150</td>
<td>11100</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>514</td>
<td>14906</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>951</td>
<td>53256</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>1429</td>
<td>157190</td>
</tr>
</tbody>
</table>

Figure 6.15: Number of generated states and transitions by selected features for Circle Nim. Each heap contained one stick; players could take from between one and three consecutive heaps.

Nine heaps and four players, 1429 states and 157190 transitions are generated, representing a 143x and 1420x increase, respectively. Only two numbers were changed in code to realize this exponential increase in output complexity.
Chapter 7

Feature-Oriented Cache Design

7.1 Introduction

Caches are ubiquitous in hardware designs. In this chapter, we combine the techniques from Chapter 5 and Chapter 6 to create a feature oriented cache. This investigation allowed us to explore feature-orientation on a much more complicated piece of hardware with orthogonal features. Most caches are either write-through or write-back, while simultaneously being either no-write-allocate or write-allocate. In order to accommodate these features, the cache must not just evolve with hardware, but also conceptually with new states and transitions. In this chapter, we develop the techniques presented earlier to create a fully synthesizable feature-oriented cache design.

7.1.1 Research Questions

In this chapter, we are examining the implications of combining techniques from the two previous chapters, so the research questions must be adapted accordingly.

To what extent can we combine modifying the generated logic and the control structures to incorporate new features? How are cross-cutting concerns handled in this situation? When applying both features on the lexical level and the semantic level, how do we ensure that both are being applied appropriately? How is the hardware generator aware of changes to the finite-state machine?
What type of characterization can we use to prepare the cache for feature application? To what extent can we leverage the type system to assist in feature application? To what extent can we isolate features into types for application?

7.2 Feature-Oriented Finite State Machines

As stated earlier, FSMs serve as the basis of control for many components of an architecture implementation, including cache control and multi-cache coherence, bus arbitration, and network protocols. We describe generally how aspects can transform such machines to implement features of interest. We then describe a library we have implemented in Scala/Chisel to simplify expression of aspects for FSMs. When run as a Chisel program, the resulting FSMs synthesize Verilog with the features of interest.

7.2.1 Writing FSM Aspects for a Cache

We have modeled the programming interface using the well established aspect language for Java, AspectJ [29]. This provides a familiar interface for aspect practitioners, as well as connects our work with prior AOP languages. No new syntax is needed to implement our library; everything is specified using ordinary Scala/Chisel.

Pointcuts  Figure 7.1 demonstrates the creation of a pointcut from our implementation. Here, the predicate is written using a Scala match statement. The Pointcutter will iterate over all the states in the FSM and add them to the pointcut if the predicate evaluates to true. A predicate can be written as any arbitrary Scala code as long as it follows the interface of consuming a state (or token) and returns true if it satisfies the properties defined in the predicate. The result of this example would create a pointcut of states, where the type of the state is WriteWaitState. This is the state in cache where it waits for acknowledgement from the backing store in the AXI protocol.

Advice  Figure 7.2 shows the construction of advice using the AfterState class. Any arbitrary Scala code can be executed inside the advice body as long as it returns advice.
val waitPointcut = Pointcutter[State, WriteWaitState](
   nfa.states,
   state => state match {
      case s: WriteWaitState => true
      case _ => false
   }
)

Figure 7.1: A pointcut from our implementation.

StateJoinpoint provides reflexive access to the join point as well as its context. In this advice, in the match statement, we test to see a transition already exists coming out of the state using the join point context. If we do not do this, the advice would apply again. The result of this advice would be to insert a new ack edge between every WriteWaitState and the Idle state. This transition represents the completion of the write transaction. Once the cache receives acknowledgement from the backing store, it returns to the idle state to wait for another transaction.

AfterState[WriteWaitState](waitPointcut, nfa)((thisJoinpoint:
   StateJoinpoint[WriteWaitState], thisNFA: NFA) => {
   thisJoinpoint.out match {
      case Some(t) => (None, thisNFA)
      case _ => (Some(WriteFSM.ack, ReadFSM.sIdle), thisNFA)
   }
})

Figure 7.2: Advice using the pointcut from Figure 7.1 from our implementation.

Hardware Generation

Our software library uses Chisel constructs to generate hardware FSMs. These are the exact same hardware constructs that Chisel uses under the hood to generate their hard-coded FSMs. Before generation, end users associate each state and token with a string ID. Then, each state is given a conditional block and the transitions placed inside with their own conditional blocks. The end user is returned a handle to the FSM. The handle has one wire associated with each state. This signal is asserted when the state becomes active. The
handle also has one assignable wire for each token. When this signal is asserted, and it is
associated with current active state, the transition occurs.

In Figure 7.3 we show how an FSM handle is used. Since the FSM interacts with the
outside world via a handle, the implementation of what happens when a state is asserted is
completely decoupled from the FSM itself. This is extremely useful from a feature-orientation
standpoint. New implementation information can be added as features are added. As long
as the FSM’s handle remains in scope and a module barrier is not crossed, the signals in the
handle can be used anywhere.

```java
1 when (fsmHandle("sReadCache")) {
2   when (hit) {
3     io.resp.valid := true.B
4   }
5 }
6
7 fsmHandle("readFinish") := !io.req.valid && hit
```

Figure 7.3: Using an FSM handle in our software library.

7.3 Feature-Oriented Cache

We next use our AOP FSM library to create a feature-oriented cache design. We present
the base machine in terms of the hooks it exposes for features, and then we define features
using those hooks. While the result is presented in its final form here, the process is one of
refactoring and evolution as new features are included.

7.3.1 Feature Decomposition

The stateful nature of the cache requires feature decomposition across both the internal
hardware and the FSM that controls it. However, the instruction and data cache designs are
not mutually exclusive. They both share the same base design for both the FSM and the
cache overall. All endpoints for either cache are the result of applying features to the same
base designs. All our designs are built for the RISC-V-mini processor [45].
We strategically chose this processor, despite its small size. A goal of feature-oriented programming is to begin with a small design and then additively build upon it. So, we chose RISC-V-mini rather than a complicated chip generator like Rocket Chip [9] to save us the time of deconstructing the fully-featured implementations already present in Rocket Chip.

Finite-State Machine

We have divided the FSM’s mechanisms into five separate features.

- **Read** provides all the functionality for a read-only cache.

- **Write** provides the functionality to write to memory. This is an abstract feature. By itself, it does not have enough to complete cache transactions. One of the two following features must also be included.

- **Acknowledge Idle** returns the cache back to the idle state after the backing store acknowledges a write.

- **Acknowledge Read** returns the cache back to the read state after the backing store acknowledges a write.

- **Dirty Accounting** provides the necessary transitions for a write-back cache.

The evolution of our FSM is shown in Figures 7.4, 7.5, and 7.6. Note, we reuse the Read feature in all of our FSMs without any modification. Furthermore, the Figure 7.5 and Figure 7.6 FSMs largely share the same structures. They share all the same states and nearly all the same transitions. Our feature-oriented approach to FSMs allows us to take the FSM in Figure 7.5 suitable for write-through and simply add new edges to make it suitable for write-back (Figure 7.6). Since the generator itself calls into our software library, all of our FSM features are subsumed into our hardware generator features (discussed in Section 7.3.1).
Hardware Generator

Below are the 9 separate hardware generation features for our cache. In all cases, the cache implements the AXI [7] interface for communication with the backing store and follows the request-response interface already present in the RISC-V-mini datapath.

To show the extensibility of our feature-oriented approach we have also included an esoteric Dusty [49, 31] feature, an idea proposed to reduce unnecessary write-backs, particularly for reference counts. When the cache is about to writing a dirty line back to memory, a dusty cache will first consult a secondary “image” of the line to see if the line has actually changed since it was first brought into cache. This is useful for fetched values that change temporarily but return soon to their originally fetched value.

- **Base System** provides the structure to which all other features can be applied.
- **HasBufferBookkeeping** enables just enough bookkeeping to hold one buffered memory transaction.
Figure 7.5: The FSM that combines Read, Write, and Acknowledge Idle.

- **HasMiddleAllocate** builds out the internal memory for the cache and allows cache lines to be allocated.

- **HasWriteStub** stubs off the write channel for read-only memory.

- **HasWriteFSM** introduces the write FSM with both the write and Acknowledge Idle features applied.

- **HasSimpleWrite** provides the hardware necessary to write to memory.

- **HasInvalidOnWrite** invalidates the buffer or cache line if the tag matches.

- **HasMiddleUpdate** allows the internal memory of the cache to be updated by writes from the datapath.
Dirty Accounting modifies the FSM with both the Acknowledge Read and Dirty Accounting features. As well as providing all the hardware necessary for handling dirty cache lines.

Dusty creates a second internal memory for the cache that acts as an image of main memory. A cache line is only considered dirty if the internal memory and image of the line are not equal.

Endpoints

As stated earlier, our features combine to form endpoints for both an instruction and data cache. Endpoints for both the instruction and data caches can exist in the same system at
the same time (how this is achieved is discussed in Section 7.3.2). This means that our cache system can realize 10 separate overall endpoints, two choices for the instruction cache and five choices for the data cache. Figure 7.7 lists all the endpoints for the instruction and data caches, as well as the features combined to achieve each of them.

The Read-Channel and Write-Channel endpoints implement the hardware necessary for memory transactions, as well as a small buffer for use with the AXI interface. In all other cases, our generator creates a direct mapped cache with 256 sets and 16 byte cache lines. Set-associative and fully-associative caches are embraced well by our approach, but they are beyond the scope of the experiments in this paper.

<table>
<thead>
<tr>
<th>Endpoint</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Cache</td>
<td></td>
</tr>
<tr>
<td>Read-Channel</td>
<td>HasWriteStub, HasBufferBookeeping</td>
</tr>
<tr>
<td>Read-Only</td>
<td>HasWriteStub, HasMiddleAllocate</td>
</tr>
<tr>
<td>Read-Only</td>
<td>HasWriteStub, HasBufferBookeeping</td>
</tr>
<tr>
<td>Write-Channel</td>
<td>HasWriteFSM, HasSimpleWrite, HasBufferBookeeping, HasInvalidOnWrite</td>
</tr>
<tr>
<td>WriteBypass</td>
<td>HasWriteFSM, HasSimpleWrite, HasMiddleAllocate, HasInvalidOnWrite</td>
</tr>
<tr>
<td>WriteThrough</td>
<td>HasWriteFSM, HasSimpleWrite, HasMiddleAllocate, HasMiddleUpdate</td>
</tr>
<tr>
<td>WriteBack</td>
<td>HasWriteFSM, HasSimpleWrite, HasMiddleAllocate, HasMiddleUpdate, Dirty Accounting</td>
</tr>
<tr>
<td>Dusty</td>
<td>HasWriteFSM, HasSimpleWrite, HasMiddleAllocate, HasMiddleUpdate, Dirty Accounting, Dusty</td>
</tr>
</tbody>
</table>

Figure 7.7: Endpoints for the Instruction and Data Caches.

7.3.2 Feature Implementation

In Chisel hardware generators such as RISC-V-mini [45], Rocket Chip [9], and BOOM [16] we have generally observed a lack of modularity in the generator code. Commonly, the organization of the generation code reflects what would usually be seen in hardware description languages, where functionality is hard-coded and entangled with other features. Despite the fact that Chisel is embedded within the Scala language, hardware designers have yet to embrace the full power of object-oriented programming and functional programming. This results in designs that are difficult to extend and reuse due to their high level of specialization. In this section, we present our feature-oriented hardware generation technique that overcomes these issues.

Our implementation combines Chisel, our library, and another aspect-oriented tool, Faust [25] that inserts code into Scala programs via abstract-syntax tree modification. We use Faust to apply all of our features automatically to the RISC-V-mini codebase. Applying each feature only takes a few seconds for Faust to modify the code base. End users can select any of the 10
design endpoints from the command line. Faust takes care of application and un-application of the features.

Figure 7.8 lists how many lines of code each it took to implement each of our features. By taking a feature-oriented approach, we were able to achieve a high level code leverage and design reuse between features. On average, each feature is only 43 lines of generation code. Our largest feature, **Dirty Accounting** is just 104 lines of code. This feature, by itself, adds all the necessary hardware to build a write-back cache out of a write-through cache.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Chisel</th>
<th>Our Library</th>
<th>Faust</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base System</td>
<td>336</td>
<td>25</td>
<td>0</td>
<td>361</td>
</tr>
<tr>
<td>HasWriteStub</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>HasWriteNFA</td>
<td>10</td>
<td>55</td>
<td>0</td>
<td>65</td>
</tr>
<tr>
<td>HasSimpleWrite</td>
<td>17</td>
<td>0</td>
<td>0</td>
<td>17</td>
</tr>
<tr>
<td>HasBufferBookeeping</td>
<td>35</td>
<td>0</td>
<td>16</td>
<td>51</td>
</tr>
<tr>
<td>HasMiddleAllocate</td>
<td>68</td>
<td>0</td>
<td>8</td>
<td>76</td>
</tr>
<tr>
<td>HasInvalidOnWrite</td>
<td>12</td>
<td>0</td>
<td>8</td>
<td>20</td>
</tr>
<tr>
<td>HasMiddleUpdate</td>
<td>21</td>
<td>0</td>
<td>8</td>
<td>29</td>
</tr>
<tr>
<td>Dirty Accounting</td>
<td>11</td>
<td>27</td>
<td>66</td>
<td>104</td>
</tr>
<tr>
<td>Dusty</td>
<td>0</td>
<td>0</td>
<td>14</td>
<td>14</td>
</tr>
</tbody>
</table>

Figure 7.8: Lines of code used to implement each feature.

**Hardware Generation Through Types**

Rather than tying the structure of the hardware generator code to the hardware hierarchy, we propose separating hardware into types with specific generation tasks. To illustrate this, the base **Cache** class is listed in Figure 7.9. Here, the **Cache** class is responsible for creating and initializing the hardware needed for the optional cache features.

Furthermore, on lines 25 and 26 we have the **Frontend** class and **Backend** class. These two classes are responsible for generating the hardware needed for communication with the datapath and the backing store, respectively. Separating these concerns and encapsulating them into their own classes, they have become decoupled from the rest of the hardware. In the future, if the design moved away from AIX to a different protocol, **Backend** could be swapped out for a different class that has the same interface, with appropriate changes to the FSM and IO for the module.

Hardware generation types give us a way to *optionally* generate hardware through class methods. On line 33 the **read()** method is called because every cache design must read from
memory. However, some endpoints require the ability to write to memory. So, the HasSimpleRead feature calls the write() method of the Frontend while the HasWriteStub feature calls writeStub(). Instead of having to write one implementation for read-only and one implementation for read-write, we can call one method or another while sharing the rest of the design. In addition, the hardware design may easily be extended by adding new hardware generation methods or overriding old ones.

Critically, hardware as types creates “hooks” for us to grab onto, to modify the abstract syntax trees with Faust. Within our generator, the instruction cache and data cache are subtypes of Cache. We direct Faust to modify the cache of a specific type with the desired features. This enables us to easily create endpoints modifying both caches without having to worry about cross contamination of features between the two.

Features as Traits

Traits in Scala function similarly to Java interfaces. Unlike Java, Scala traits support multiple inheritance and Scala code can be called from within a trait. Packaging functionality into traits is not a new idea. RISC-V-mini, Rocket Chip, and BOOM all have traits that implement some functionality. However, our approach differs significantly in that we package whole features into traits. Thus, to add a feature to a design, the type can just be extended with the trait containing the feature.

Consider the HasMiddleAllocate feature in Figure 7.10. By extending the InstructionCache with HasMiddleUpdate instead of HasBufferBookeeping the read-channel is transformed into a read-only cache. By combining this with our type encapsulation technique (Section 7.3.2), zero hand rewriting of the design is required to make this extensive change.

Since Faust modifies the abstract syntax tree of the hardware generator, creating different endpoints only requires instructing Faust to extend the cache classes with different traits. All of our endpoints except for WriteBack and Dusty are created this way.

Inserting Hardware into Traits
Sometimes, there are features that cannot be succinctly captured in a single trait, but instead crosscut the hardware generator. To implement a Write-Back cache, new features must be added to the FSM, hardware must be created for storing the dirty bit and transmitting that information, as well as creating or changing conditions for cache updates and memory transactions.

In this instance, we heavily utilize the aspectual nature of Faust. The majority of the implementation of the implementation of Dirty Accounting is written as an aspect in Faust (see Figure 7.8). However, Dirty Accounting requires more than just the extension of traits with new traits. We use Faust’s ability to weave new code into the hardware generator to modify and extend the designs contained in traits. HasMiddleUpdate is modified to hold the dirty bits, as well as connect with the new mechanisms in the FSM for write-back. HasWriteFSM receives new features to account for dirty cache lines and writing back to memory. HasSimpleWrite is updated with new conditions for writing to memory. Conveniently, this can all be contained in one selectable aspect within Faust.

This is further exemplified by the Dusty feature. The whole implementation of Dusty is captured in a single aspect within Faust. Figure 7.11 shows the relatively small amount of work needed to implement Dusty. First, once again utilizing the type isolation technique (Section 7.3.2), we create our reference image of memory by instantiating another Middleend class. Then, we provide advice around the dirty signal to judge a cache line dirty only if it does not equal what exists in the reference image of memory.

7.3.3 SIMD Caches

Ultimately, the goal of this work is to make the design of complex hardware easier through generation. Here, we will demonstrate how we can combine our techniques with “off-the-shelf” components to generate SIMD Caches that are coherent with each other. While we recognize that this is purely a pedagogical example, it is modeled off the real-world RDNA Architecture from AMD [26].

The RDNA architecture packages two SIMD execution units into a single unit called a workgroup of processors. Each workgroup shares L0 cache between each SIMD unit. This cache is kept coherent through serialization of the execution when conflicts are detected.
Two workgroups are then packaged together into a *Dual Compute Unit* (DCU). Currently, the two L0s within a DCU are kept coherent via software.

Suppose that we wanted to model a similar cache system ourselves, but instead of handling data conflicts between the two SIMD units in a workgroup processor, we use the MSI cache coherence protocol [32]. For this case study, we have selected the ready-made cache from RISC-V Mini [45]. As-is, this cache, shown in Figure 7.12, is directly connected to main memory in a single core system and does not have any coherence protocol. Using a feature-oriented approach, we can retrofit this cache with advice that implements the MSI protocol, shown in Figure 7.13. In order to model the interactions between the two now coherent caches, we can cross-product the two. This results in an FSM with 729 states and 2,461,104 transitions. Hardware designers could use this sort of modeling for correctness verification, however that is beyond the scope of this paper.

Imagine that in the next iteration of the architecture, the designers wanted to use hardware coherence between all the SIMD units in a DCU. Instead of having to start the model completely from scratch, hardware designers can simply instantiate two more caches into the system. Furthermore, rather than being locked into a single cache coherence protocol, the hardware designers may want to choose the MESI protocol [60], for their next iteration. With our approach, this protocol can just be applied instead of the MSI protocol.

### 7.4 Performance and Area

We have taken our feature-oriented designs all the way up through simulation and synthesis. The hardware generator is implemented in Chisel 3.5.1. All designs were emitted as Verilog. Beyond the cache, the rest of the RISC-V-mini was kept original, implementing the RV32I ISA of the User-level Version 2.0 [75] and the Machine-level ISA of the Privileged Architecture Version 1.7 [76]. We simulated the large benchmarks from the RISC-V tests repository [63] using Verilator 4.214 [74]. Designs were synthesized for an xc7a100tcs324-1 FPGA using Vivado 2022.1 with the Vivado synthesis defaults. Figure 7.14 shows the cycles per instruction for each benchmark, with the average CPI of all the benchmarks displayed at the bottom. Figure 7.15 shows the synthesized area of the whole synthesized chip design in LUTs.
The smallest endpoint, readChannel-writeChannel takes only 55.2% of the area of our largest endpoint, readOnly-dusty, which reduces CPI by 47.5%. By feature-orienting our design, we have enabled a fine grain of design space exploration. We can see the CPI drop nearly linearly as more features are added to the design. Coarse grained analysis is not lost in this technique either. Comparing the two instruction cache endpoint groups, we see that the addition of an instruction cache has an average decrease in CPI of 1.19 cycles and an average increase in LUTs of 438.

While this result is not surprising, adding an instruction cache should improve performance at the cost of increasing design area, we speculate that this technique can help to quickly identify the properties of new designs and save designers time. For instance, the readOnly-writeChanel has a lower CPI than any of the readChannel endpoints and takes less area than all of them except readChannel-writeChannel. This analysis tells us going down the path of more advanced data cache features is not worth the area cost if there is no data cache. Again, this is not surprising, but we posit that there are other opportunities in hardware designs for this sort of analysis.

A feature-oriented approach can help hardware designers better balance trade-offs while maintaining high levels of design reuse. Without the need to start hardware designs from scratch, designers can accomplish quicker prototyping while maintaining a high level of analysis and code reuse.
class Cache(val c: CacheConfig, val nasti: NastiBundleParameters, val xlen : Int) extends Module {
  // outward facing IO
  val cpu = IO(new CacheIO(xlen, xlen))
  val mainMem = IO(new NastiBundle(nasti))

  // local parameters
  protected val p = new CacheParams(c.nSets, c.blockBytes, xlen, nasti)

  // build the NFA for the cache
  protected lazy val cacheNFA = Weaver[NFA](List(), ReadFSM(), (before: NFA, after: NFA) => before.isEqual(after))
  protected lazy val fsmHandle = ChiselFSMBuilder(cacheNFA)

  // this section is for things needed by the frontend and the backend
  // split the address up into the parts we need
  protected val nextAddress = Wire(chiselTypeOf(cpu.req.bits.addr))
  protected val address = Reg(chiselTypeOf(nextAddress))

  // create a buffer for reads from main memory, also store the tag
  protected val buffer = Reg(Vec(p.dataBeats, UInt(nasti.dataBits.W)))
  // we will always have at least this
  val valids = RegInit(0.U(p.nSets.W))

  // set up phases used in every cache
  protected val front = new Frontend(fsmHandle, p, cpu)
  protected val back = new Backend(fsmHandle, p, mainMem, address)

  // hit can be several different things depending on the feature set
  protected val hit = Wire(Bool())
  hit := false.B

  front.read(hit)
  ...
trait HasMiddleAllocate extends Cache {
  val middle = new Middleend(fsmHandle, p, address, tag, index, valids)
  val (oldTag, readData) = middle.read(buffer, nextAddress, offset, Some(hit), Some(cpu))
  middle.allocate(Cat(mainMem.r.bits.data, Cat(buffer.init.reverse)), readDone)
}

Figure 7.10: HasMiddleAllocate feature.

class Dusty extends Feature {
  before(q"val isDirty = dirty(index)") insert (q""
    val dusty = new Middleend(fsmHandle, p, address, tag, index, valids)
    val (_, dustyData) = dusty.read(buffer, nextAddress, offset)
    dusty.allocate(Cat(mainMem.r.bits.data, Cat(buffer.init.reverse)), readDone)
  "") in (q"trait HasMiddleUpdate") register

  around(q"val isDirty = dirty(index)") insert (q"val isDirty = dirty(index) && (dustyData /= readData)"
    in (q"trait HasMiddleUpdate") register
}

Figure 7.11: Dusty feature in Faust.
Figure 7.12: The cache FSM from the RISC-V Mini.
Figure 7.13: The cache FSM from RISC-V Mini with the MSI protocol.
<table>
<thead>
<tr>
<th>benchmark</th>
<th>Write-Channel</th>
<th>Write Bypass</th>
<th>Write Through</th>
<th>Write Back</th>
</tr>
</thead>
<tbody>
<tr>
<td>median</td>
<td>3.59</td>
<td>3.11</td>
<td>3.11</td>
<td>3.02</td>
</tr>
<tr>
<td>multiply</td>
<td>2.81</td>
<td>2.78</td>
<td>2.78</td>
<td>2.77</td>
</tr>
<tr>
<td>qsort</td>
<td>3.48</td>
<td>3.38</td>
<td>3.19</td>
<td>3.00</td>
</tr>
<tr>
<td>towers</td>
<td>3.78</td>
<td>3.69</td>
<td>3.36</td>
<td>2.46</td>
</tr>
<tr>
<td>vvadd</td>
<td>3.71</td>
<td>3.07</td>
<td>3.07</td>
<td>3.00</td>
</tr>
<tr>
<td>Average CPI</td>
<td>3.47</td>
<td>3.21</td>
<td>3.10</td>
<td>2.85</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>benchmark</th>
<th>Write-Channel</th>
<th>Write Bypass</th>
<th>Write Through</th>
<th>Write Back</th>
</tr>
</thead>
<tbody>
<tr>
<td>median</td>
<td>2.41</td>
<td>1.93</td>
<td>1.93</td>
<td>1.87</td>
</tr>
<tr>
<td>multiply</td>
<td>1.37</td>
<td>1.33</td>
<td>1.33</td>
<td>1.33</td>
</tr>
<tr>
<td>qsort</td>
<td>2.11</td>
<td>2.01</td>
<td>1.83</td>
<td>1.64</td>
</tr>
<tr>
<td>towers</td>
<td>2.84</td>
<td>2.76</td>
<td>2.42</td>
<td>1.61</td>
</tr>
<tr>
<td>vvadd</td>
<td>2.64</td>
<td>2.00</td>
<td>2.00</td>
<td>1.93</td>
</tr>
<tr>
<td>Average CPI</td>
<td>2.27</td>
<td>2.01</td>
<td>1.90</td>
<td>1.68</td>
</tr>
</tbody>
</table>

Figure 7.14: Cycles per instruction for each RISC-V benchmark, by endpoint.
Figure 7.15: The area in LUTs of the endpoints.
Chapter 8

Conclusions and Future Work

In this dissertation, we have presented a foray of feature-oriented hardware design. By generalizing hardware specifications through the mediums of adders, performance counters, and caches, we have shown that a larger design space, both qualitatively and quantitatively, is achievable through feature-oriented design.

Ultimately, we hope this work inspires others to adopt a feature-oriented mindset when designing hardware. We envision open hardware characterizations where designers in both research and industry can share, test, and compare designs freely. This could open up a marketplace for hardware features, allowing the best design to be chosen among many for their applications.

In support of this goal, we have started work on addressing some shortcomings of the work presented in this dissertation. Namely, the largest weakness of this work is the over reliance on the specific software characterization of hardware generators. We have relied on “hooks” that come directly from code statements rather than something more generalizable.

8.1 Faust 2

To resolve this, we have started work on “Faust 2”. In this complete re-implementation, we rely not on the specific text of the hardware generator, but the type structure of the code. By expressing join points through types, anyone who implements these types can utilize the aspects that hook into them.

Figures 8.1 and 8.2 demonstrate hooking into types. In Faust 2, pointcuts are constructed based on method calls. Advice will be applied at all locations where that method is called, unless otherwise specified by a type restriction. The advice body always returns the value of
the join point, requiring the return type to be specified. We have achieved this by bringing Faust in line with traditional aspect compilers [36] by employing a join point shadow in the background. This join point shadow automatically constructs code that will capture the original return value and hand it back to the caller.

```scala
class SimpleCounter extends Aspect {
  After(("Cache", "front.read"), "Any") { (hit: Bool) => {
    val event = if (this.isInstanceOf[mini.cache.ICache]) ("iCacheHit", 0.U) else ("dCacheHit", 1.U)
    mini.cache.CacheEvents(event._2, event._1, hit)
  }
}
```

Figure 8.1: A simple cache hit event in Faust 2.

```scala
class PlaceCounters(classLoc: String, func: String, ret: String = "Any")
  extends Aspect {
  After((classLoc, func), ret) {() => {
    import mini.counter._
    val perfCounters = PerformanceCounters(Set(EventSet)(mini.cache.CacheEvents), 4)
  }}
}
```

Figure 8.2: An aspect that aggregates events and creates performance counters.

We have also made a few quality of life improvements in Faust 2 that further deepen integration into the Scala type system. Instead of requiring quasiquotes, we utilize Scala Macros [66] to allow users to create advice in plain Scala code. Thus, advice is fully type-checked by the Scala compiler. The immediate environment of the join point can also be accessed by creating matching parameters in the advice function. Line 2 in Figure 8.1 takes in the “hit” signal from the surrounding cache the advice will be applied within. Furthermore, the proceed() keyword is now supported and will automatically be replaced with the join point.

### 8.2 Hardware Types

A running theme in this dissertation has been the need for encapsulation of hardware into types and generation task. The refactoring of various characterizations [45, 9, 16] has enabled
us to apply features into these implementations. As a part of Faust 2, we also completely re-implemented a performance counter infrastructure and have completely decoupled it from both current characterizations of RISC-V and RISC-V itself. We achieved this by combining the Chisel BoringUtil and hardware types. Chisel BoringUtil [17] allows a source signal to “bore” through the hardware hierarchy to its sink. This eliminated the need to create any advice for event infrastructure. By encapsulating the performance counters into a Scala object, all that is needed to create and capture events is calling the apply method of the object anywhere in the hardware generator. For access through the architecture, they can be placed deep within the hardware hierarchy. It could be dropped into the top level for access through external monitoring hardware. The counters could even be placed in both locations.

In future work, an entire type scheme for a whole chip could be created. This could lay the basis of an entire feature oriented framework for chip design. Then any hardware implementation that adheres to this type system could be applied. This would also create a robust set of join points for cross-cutting features.

8.3 Performance Aware Hardware

In various chapters, we have reported space and performance for various design endpoints generated using our technique. In future work, this information could be fed back into the system to automatically optimize the design for a given set of constraints. Hypothetically, the feature-oriented performance counter system could be integrated during the design-test phase of optimization, in the same way that breakpoints and instrumentation are added to programs to find bugs and performance bottlenecks. Information, such as CPI, cache misses, etc. could be used to make drop-in design choices for our feature-oriented cache as well as any future feature-oriented hardware. After design, the system could be easily eliminated to save space.
case class Event(id: UInt, name: String)

abstract class EventSet(val id: UInt) {
  val events = scala.collection.mutable.Set[Event]()

  def apply(id: UInt, name: String, signal: Bool) = {
    BoringUtils.addSource(signal, name)
    events += Event(id, name)
  }
}

object PerformanceCounters {
  def apply(eventSets: Set[EventSet], numCounters: Int): Array[(UInt, mini.counter.Counter)] = {
    val signalArray = VecInit.fill(eventSets.size, 32)(Reg(Bool()))

    eventSets.foreach(es => {
      es.events.foreach(e => {
        val eventSignal = WireInit(false.B)
        BoringUtils.addSink(eventSignal, e.name) // sink to the id bit
        signalArray(es.id)(e.id) := eventSignal
      })
    })

    // build out the counters
    Array.fill(numCounters) {
      val configReg = RegInit(0.U(64.W))
      val counter = new mini.counter.Counter() with OnRisingEdge

      // connect to the correct event set
      val eventVec = signalArray(configReg(63, 32)) // use the upper 32 bits for event set
      // count only when all configured events have triggered
      counter.count((eventVec.asUInt & configReg(31, 0)) === configReg(31, 0)) // use lower 32 bits for individual events
      (configReg, counter)
    }
  }
}

Figure 8.3: Re-implementation of the performance counter system.
References


88


Appendix A

Foam Library Manual

In this section, we will do a deeper dive into the finite-state machine library we implemented briefly mentioned in Chapter 6. This work was completed with the assistance of Peyton Gozon, and Max Camp-Oberhauser. Our library is complete written within Chisel and is available on GitHub.

From a high level, programmers describe their FSMs using constructs within the software library that we provide. Users can then define aspects by extending the already existing feature classes that contain methods to create pointcuts and advice for either states or transitions. Finally, resulting FSM may be integrated into any Chisel program through the library.

Since the library is implemented using standard Scala, any legal Scala syntax can be used to manipulate and extend our library. This provides end users with a powerful tool to reason about the construction and properties of their FSMs directly through the software interface.

To guide this Chapter, we will continue use a vending machine as primary example.

A.1 Building a Base FSM

Every FSM implementation in Foam begins by extending the NFA Class. Initial states and transitions can be added using the addTransition method that accepts a state-transition tuple and a destination state. The library automatically incorporates new tokens and states without user intervention. Our library upholds immutability its component parts, so all transformations of the NFA result in a new NFA object. Figure A.1 shows the two lines needed to initialize our Vending Machine FSM.
All states and tokens must extend the `state` or `token` traits. Foam can operate with any legal sub-type of these two traits. Basing states and tokens on types allows us to combine many different attributes and represent and reason about complex ideas within the generation and FSM itself. Since we operate on NFAs, we do provide a `Lambda` token for moving ahead without consuming input. For example, our vending machine has four state types and three token types that extend our base traits with one or more of five other traits.

- A `ValueState` represents some amount of currency that has been entered into the machine. All other state types inherit from this.
- `DispenseState` signifies that a selected product should be dispensed from the system.
- The `PrinterState` tells the machine that some message must be presented to the user.
- `ChangeState` provides the change contained in the vending machine back to the user.
- The `Coin` token represents a currency of some value.
- Users select a product with the `Product` token.
- Any general activity that the vending machine would represent, such as returning change, can be implemented with the `System` token.

### A.2 Adding Features

Within Foam, we implement features via *aspects*. Each aspect must extend the `Aspect` trait and define the `apply` method to take in an FSM and produce a new FSM. Foam chains aspects together, taking the output of one and feeding it into the input of another, to produce an FSM. This process repeats until the FSM stops changing. Unlike aspects in AspectJ that takes in parameters *from the joinpoint*, we allow direct parameterization of the aspect.
The type signature in Figure A.2 contains two parameters, `coin` and `threshold`. Coin denotes the denomination to be added to the FSM and threshold controls the maximum value the vending machine can contain. Thus, this aspect can be reused for any denomination–threshold combination. One consequence of our aspects being plain Scala classes is that any arbitrary Scala code can be used within the definition, as long as the aspect interface is respected.

```
1 class AddCoin(coin: Coin, threshold: Int) extends Aspect[NFA] {
2    ...
3 }
```

Figure A.2: The type signature of the Add Coin class.

### A.2.1 Building a Pointcut

Foam can compose pointcuts out of either objects that extend the `state` or `token` traits, including their sub-types. We provide a `Pointcutter` utility to assist in the creation of pointcuts. The `pointcutter` takes in a set of states or tokens, then filters them based on the provided predicate. It returns a `Pointcut` object with the appropriate parametric type of the elements in the pointcut. Every item in the pointcut must be the same type, so they must inherit a common super class.

Foam users can use any function that implements their predicate as long as the function takes in an element of the set and returns a Boolean value. Since, this too is plain Scala code, any outward facing properties of the objects can be used to determine inclusion. In Figure A.3 the predicate is conditioned on whether the state is of type total state and if the state’s value in addition to the value we want to add, is greater than or equal to the threshold.

Conceptually, the vending machine cannot accept coins while it is performing another activity. A vending machine would not allow depositing more coins while giving back change or vending a product. This predicate prevents spurious transitions from states where coins should not be inserted. The predicate also ensures termination of feature application. By placing a threshold on the pointcut by examining the values they characterize we know that the highest valued states will not be included.

[95]
A.2.2 Advice

Like the aspect itself, each advice object takes in an FSM and produces a new one. Foam does this by iterating over the joinpoints in the pointcut and applying the advice, producing a new FSM each time. End users may chain advice application inside of an aspect by passing the resulting FSM from one advice object to another. To support this, every advice object requires three parameters: a pointcut, an FSM, and a implementation function. The implementation function provides Foam with the necessary information to apply the advice on the joinpoint.

Figure A.4 shows the function signature for the AfterState implementation function. The function is parameterized by the current joinpoint and the current iteration of the FSM. The current joinpoint parameter allows reflective access to the current joinpoint. The end user implements the body of the function to return new state and token information and the FSM the advice will be applied to. The implementation function can be any arbitrary Scala code, so long as the function signature is respected. Since the end user may produce a new FSM inside of the implementation function, we require that the FSM be passed back from the implementation function.

Everything described above comes together in Figure A.5 showing the full definition of the AddCoin class. The AroundState object takes in the statePointCut and the input NFA for the aspect. In the body of the implementation function, the reflective access to the joinpoint (thisJoinPoint) is used to create a new transition using the new coin token from the joinpoint to a TotalState that represents the increased total in the vending machine. The function then returns the joinpoint and resulting FSM for the next iteration.

Figure A.3: The predicate for the Add Coin class.

Figure A.4: Function signature for the implementation function for advice.
class AddCoin (coin: Coin, threshold: Int) extends Aspect[NFA] {
  def apply (nfa: NFA) = {
    // select all the states of type TotalState
    val statePointCut = Pointcutter[State, TotalState](nfa.states, state =
      state match {
        case s: TotalState if(s.value + coin.value <= threshold) => true
        case _ => false
      })
    AroundState[TotalState](statePointCut, nfa)((thisJoinPoint: Joinpoint[TotalState], thisNFA: NFA) => {
      val newNFA = thisNFA.addTransition((thisJoinPoint.point, coin),
        TotalState(thisJoinPoint.point.value + coin.value, true))
      (thisJoinPoint.point, newNFA)
    })
  }
}

Figure A.5: The full AddCoin class.

Advice patterns  We characterize the state and token information needed by Foam for advice application as patterns. These are alternating patterns of the state and token that would be inserted into the FSM. Currently, Foam supports advice patterns of length one or two, but there is theoretical limitation on the length of these patterns. Advice patterns are broken into three categories: single state or token, state-token, or token-state.

- **Single State or Token**: This pattern is used by the AroundState and AroundToken advice objects respectively. Naturally, to perform around advice on a state or token only a single replacement must be specified.

- **State-Token**: This pattern is used by the BeforeState and AfterToken objects. BeforeState will insert advice in between all the transitions into the state and the state itself. The advice state in this pattern will take the place of the joinpoint and the advice token will provide the new transition from the advice state to the joinpoint. AfterToken inserts its advice between the transitions using the token and their destination states. The advice state takes the place of the old destination and the advice token provides the new transition from the new destination to the old destination state.

- **Token-State**: This pattern is used by the AfterState and BeforeToken objects. AfterState inserts advice information between the joinpoint and all the transitions
out of the joinpoint. The advice token provides the new transition to the advice state and the advice state becomes the new source for the transitions out of the joinpoint. **BeforeToken** inserts advice information in between every state and transition containing the token. The token advice provides a transition to the advice token, the advice state is then the source of the joinpoint transition.

**Context aware advice** Beyond reflective access to the joinpoint, Foam also provides reflective access to the context of each joinpoint. A state joinpoint’s context can be accessed by parameterizing the implementation function with the StateJoinpoint class instead of Joinpoint. The state joinpoint context takes the form of either a state-token pattern, for every transition into the state, or a token-state pattern, for every transition out of the state. Similarly, a token joinpoint’s context can be accessed through the TokenJoinpoint class. In this case, the advice receives the states on either end of every transition the token is in. Foam applies advice not only for every joinpoint, but also every joinpoint context. Thus, advice can be dependant not only on the particular joinpoint, but each context as well.

Figure A.6 demonstrates this technique. Here we want to add a warning about peanut content **before** that item can be selected for purchase. The pointcut contains all the states that dispense and item with peanuts in them. We ascertain the source of the transition into the peanut state through the context. If the source was already a peanut warning, no advice is applied. If not, then a warning is inserted from the context source state to the warning state and a transition on “accept” is created to the state dispensing the item containing peanuts.

### A.3 NFA to DFA Conversion

Foam also supports converting non-deterministic finite automata into deterministic finite automata. This conversion happens automatically when a new DFA class is instantiated. The DFA class is parameterized with an NFA object and an error state. The resulting object contains the DFA. In the conversion to a DFA, some states and tokens may be combined. In traditional finite state machines, there is no issue with this combination. For FSMs for hardware, we need to maintain the original order that states occurred in to not upset the order of control. In the DFA, Foam has two special types of state and tokens
BeforeState[ValueState](pointcut, nfa)((thisJoinPoint: StateJoinpoint[ValueState], thisNFA: NFA) => {
  val source = thisJoinPoint.in.get._1.asInstanceOf[ValueState]
  val warning = PrinterState(warningString, source.value, false)

  source match {
    case s: PrinterState if(s.action == warningString) => (None, thisNFA)
    case _ => {
      val newNFA = thisNFA.addTransition((warning, System("Reject")), source)
      (Some(warning, System("Accept")), newNFA)
    }
  }
})

Figure A.6: Advice used by Peanut Warning to add a new warning to the FSM.

called MultiState and MultiToken. When executing these states or tokens, the order of the original component states and tokens are preserved while still ensuring that the finite automata is deterministic.