Tampere University

Kari Hepola

GENERATION OF CUSTOMIZED RISC-V IMPLEMENTATIONS

Master of Science Thesis
Faculty of Information Technology and Communication Sciences
Examiners: D.Sc. Joonas Multanen
Prof. Pekka Jääskeläinen
January 2022

ABSTRACT

Kari Hepola: Generation of Customized RISC-V Implementations
Master of Science Thesis
Tampere University
Master's Degree Programme in Electrical Engineering
January 2022

Processor customization has become increasingly important for achieving better performance and energy efficiency in embedded systems. However, customizing processors is time-consuming and error-prone work. The design effort is reduced by describing the processor architecture with high-level languages that are then used to generate the processor implementation. In addition to processor customization, open source hardware and standardization have become increasingly more popular. RISC-V that is a relatively new open standard instruction set architecture, has gained traction both in academia and industry.

This thesis work added a RISC-V extension to the OpenASIP toolset that is developed at Tampere University. OpenASIP has wide support for customizing and generating transport triggered architectures. Transport triggered architectures have an exposed datapath that is visible to the programmer, which allows a lower level programming interface. The hardware generation and customization features in OpenASIP were reused by utilizing a transport triggered architecture as the internal microarchitecture together with a microcode unit. The extension generates the RISC-V implementations from an architecture description, which reduces the design effort of customizing the implementation.

The RISC-V generator developed in this thesis has customization points for the bypass network, amount of pipeline stages, operation latencies and an optional addition of the standard M extension. The generator was evaluated by generating RISC-V cores with different customization points and comparing their performance and post-synthesis properties with open source implementations. The generated cores with bypass network achieved better performance while consuming slightly more area than the smallest reference design. The microcode hardware only utilized 3.6% of the design area and did not affect the maximum clock frequency.

Keywords: RISC-V, TTA, processor customization, ASIP

TIIVISTELMÄ

Kari Hepola: Räätälöityjen RISC-V-toteutuksien generoiminen
Diplomityö
Tampereen yliopisto
Sähkötekniikan DI-ohjelma
Tammikuu 2022

Prosessorien räätälöinnistä on tullut yhä tärkeämpää sulautettujen järjestelmien suorituskyvyn ja energiatehokkuuden lisäämisessä. Prosessorien räätälöinti on kuitenkin työläs ja virhealtis prosessi, jonka työmäärää voidaan keventää kuvaamalla prosessorin arkkitehtuuri korkean tason kielillä, joita käytetään prosessoritoteutuksen generoimisessa. Prosessorien räätälöinnin lisäksi avoimen lähdekoodin laitteisto ja standardointi ovat kasvattaneet suosiotaan. RISC-V on verrattain uusi avoimen standardin käskykanta-arkkitehtuuri, joka on saanut suosiota sekä akateemisessa maailmassa että teollisuudessa.

Tässä diplomityössä lisättiin RISC-V-laajennos Tampereen yliopistossa kehitettäviin OpenASIP-työkaluihin. OpenASIP-työkaluissa käytettävä siirtoliipaisuarkkitehtuuri on prosessorisuunnittelufilosofia, jossa suorittimen datapolku on avoin ohjelmoijalle, mikä mahdollistaa matalamman tason ohjelmointirajapinnan. OpenASIP-työkalujen ominaisuuksia uudelleenkäytettiin hyödyntämällä siirtoliipaisuarkkitehtuuria suorittimen sisäisenä mikroarkkitehtuurina ja lisämäällä siihen mikrokoodiyksikkö. Laajennos generoi RISC-V-toteutukset arkkitehtuurikuvauksesta, mikä vähentää räätälöintiin liittyvää työmäärää.

Diplomityössä toteutulla RISC-V-generaattorilla voi räätälöidä prosessorin liukuhihnatasojen määrää, rekisteripankin ohituskytkentöjä ja lisätä RISC-V-spesifikaatiossa määritellyn M-laajennoksen. Generaattoria arvioitiin vertaamalla eri räätälöintivalinnoilla generoitujen prosessorien suorituskykyä ja synteesin jälkeisiä ominaisuuksia avoimen lähdekoodin toteutuksia vastaan. Generoidut prosessorit, joissa oli rekisteripankin ohituskytkennät saavuttivat parhaimmat suorituskykytulokset ja kuluttivat vain lievästi enemmän pinta-alaa kuin pienin verrokkitoteutus. Mikrokoodiyksikkö kulutti vain 3,6% ytimen pinta-alasta eikä vaikuttanut maksimikellotaajuuteen.

Avainsanat: RISC-V, TTA, prosessorin räätälöinti, ASIP

PREFACE

The work in this thesis was carried out in the Customized Parallel Computing research group at Tampere University. This project has received funding from the ECSEL Joint Undertaking (JU) under grant agreement No 783162 (FitOptiVis). The JU receives support from the European Union's Horizon 2020 research and innovation programme and Netherlands, Czech Republic, Finland, Spain, Italy. The project was also supported by European Union's Horizon 2020 research and innovation programme under Grant Agreement No 871738 (CPSoSaware) and Academy of Finland (decision #331344).

I would like to thank my supervisors D.Sc. Joonas Multanen and Prof. Pekka Jääskeläinen for making this thesis work possible and for the excellent feedback and guidance I received along the project. I would also like to thank my coworkers in the group for creating a fun and positive work environment. Last but not least, I would like to thank family and friends for supporting me during my studies.

In Tampere, Finland, 18th January 2022

Kari Hepola

CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

ADF	Architecture Definition File
ADL	Architecture Description Language
ALU	Arithmetic Logic Unit
ASIC	Application-Specific Integrated Circuit
ASIP	Application-Specific Instruction-set Processor
AUIPC	Add Upper Immediate to Program Counter
CISC	Complex Instruction Set Computer
CU	Control Unit
DAG	Directed Acyclic Graph
GPP	General Purpose Processor
IDF	Implementation Definition File
ILP	Instruction-Level Parallelism
IPC	Instructions Per Cycle
ISA	Instruction Set Architecture
JAL	Jump and Link
JALR	Jump and Link Register
LSU	Load-Store Unit
OSAL	Operation Set Abstraction Layer
RISC	Reduced Instruction Set Computer
RTL	Register Transfer Level
SoC	System-on-Chip
TTA	Transport Triggered Architecture
VLIW	Very Long Instruction Word

1. INTRODUCTION

Processors are important components in digital systems. Rising performance and energy efficiency requirements have created motivation for more optimized processors. One way to achieve a better quality of results is to tailor the processor heavily to the targeted use-case. Processors are, however, complex systems, and customizing them manually in a *register transfer level* (RTL) description is a time-consuming and error-prone task. The effort required for processor customization is decreased when the processor architecture is specified on a higher-level architecture description that is used to automatically produce the synthesizable RTL.

The complexity of designing processors is not the only challenge hindering innovation in processor design. Commercial *instruction set architectures* (ISA) have restricted the implementation of processors because of their proprietary nature. RISC-V is a relatively new ISA that has gained traction both in academia and industry due to its open-standard nature. Besides being open-standard, RISC-V is a suitable candidate for *application-specific instruction-set processors* (ASIP) because of the option for adding custom instructions and the optional standard extensions that create a modular structure for the ISA.

In this thesis, a RISC-V extension was added to the OpenASIP toolset, which uses an architecture description format to generate customized RISC-V implementations with different amount of pipeline stages, operation latencies, standard extensions and bypass connectivity. The generator uses a *transport triggered architecture* (TTA) as the internal *microarchitecture* together with a generated *microcode* layer to implement the RISC-V ISA. The microcode hardware consists of lookup tables that translate RISC-V instructions to micro-operations that are sequenced separately. The microcode layer is essentially a design-time RISC-V front end that allows the reuse of hardware generation features that are found in OpenASIP. In this work, microprogramming is purely a method for implementing part of the control and decode logic for a RISC-V core.

The structure of the thesis is divided into the following chapters. Chapter 2 introduces different processor design philosophies, instruction set architectures, as well as ways to exploit instruction-level parallelism in processor designs. Chapter 3 gives an overview of processor customization by exploring architecture description languages, the OpenASIP toolset and available RISC-V generators. Chapter 4 describes the implementation of the microcode hardware, its function in the processor pipeline and its integration into the OpenASIP toolset. Chapter 5 evaluates the performance and synthesis results of generated RISC-V cores as well as the ways the hardware was verified. Chapter 6 discusses the ideas for future work and the ways they could be implemented. Chapter 7 concludes the thesis.

2. PROCESSORS

Processors are complex programmable hardware components that perform computation on external data. In order to program processors, the programmer must have information about the processor architecture. This architectural information is described in the *instruction set architecture* (ISA), such as ARM or x86. An ISA does not describe the internal microarchitecture of a processor implementation, but only the details that are needed to program the processor. This chapter focuses on the processor design philosophies and explores different example architectures.

2.1 Complex Instruction Set Computers

Complex instruction set computer (CISC) is an ISA design philosophy. Like in the name, this design philosophy makes use of so called complex instructions. CISC instructions execute long sequences of basic operations in their instructions, even processing data that is in the memory. An example of this is loading values from memory, doing an arithmetic operation and storing the result back to memory all in one instruction. A typical CISC instruction set has both register-to-register, register-to-memory and memory-to-memory operations, which causes multiple addressing modes. This causes an issue because when the operand described in the instruction word is in memory, it takes many bits to express the memory address. To support different amount of operands that can be either in memory or in the internal registers, the instruction words can be variable length, which complicates the instruction decoding and scheduling. [1] The most popular CISC ISA is the x86 that has variable length instructions that range from one to seventeen bytes [2].

Historically, there were many motivations for complex instructions, most of which were caused by memory constraints. Complex instructions meant that fewer instructions would have to be executed in total, which resulted in better code density. This added motivation for the use of complex instructions in early computers, as memory was expensive. Additional constraint was the speed gap between memory and the processor core, which added motivation for higher-level instructions to improve the performance of the system. [3] In addition to the hardware properties, complex instructions were used to close the semantic gap between high-level languages and hardware [1].

The downside of CISC implementations is the complex control hardware that enables the execution of long sequences of basic operations described in CISC instructions. To simplify the control unit design, CISC implementations use a method called microprogramming. In this method, the control unit of the processor core is embedded with *microcode* that translates a complex instruction to a sequence of simpler *micro-operations*. [1] Microprogramming is not a new concept for creating control units as it was already proposed in the 1950's by M.V Wilkes [4].

2.2 Reduced Instruction Set Computers

Another ISA design philosophy is the *reduced instruction set computer* (RISC). The most popular RISC-based ISA family is ARM that is dominant in embedded computing [2]. In the RISC philosophy, the instruction set of a processor consists of simple operations compared to the complex instruction set computer philosophy, where single instructions can execute a long sequence of basic operations. RISC systems usually follow the load-store architecture, where only separate load and store operations move data between the memory and the register file and other instructions don't operate directly with operands that are stored in the memory. [1]

In load-store architectures the amount of addressing modes is reduced because only the separate load and store operations access memory. This allows ISAs to more easily design fixed-length instruction formats. Besides the fixed length, the formats can more easily use fixed boundaries in the subfields of the instruction word that simplifies the decoding of instructions. The control logic in RISC implementations can be easily constructed with hardwired logic without the use of microcode because of the more simple semantics of the instructions. [1]

2.3 Pipelining and Hazards

Pipelining is a common way to optimize the speed of execution in processor implementations. It works by splitting the processor core to multiple stages of execution and feeding the pipeline a new instruction each cycle. This way, the *critical path* of the core is shorter and the core can achieve a higher clock frequency. [2] Pipelining works similarly as an assembly line in a factory; the manufacturing of the product is divided into multiple steps that are done in a sequence. Multiple workers can then each do their own step in the assembly line and this way higher throughput can be achieved.

An example of a classic 5-stage RISC pipeline is presented in Figure 2.1. As seen in the figure, the core is divided into five pipeline stages: instruction fetch, instruction decode, execute, memory access and writeback. The instruction fetch is responsible for fetching a new instruction from the instruction memory. Part of the essential control flow functionality is done in this stage, as it includes the program counter register that stores the address of the fetched instruction. In the next stage, instruction decode, the instruction is decoded. Essentially, this step transforms the bits from the instruction word to control signals in the hardware. Traditionally, the core's register file is read in this step as well. In the execute stage, the *arithmetic logic unit* (ALU) is used to perform an arithmetic or logical operation. The executed operation is controlled by the control signals that were decoded in the previous stage. In the described pipeline, the ALU is also used to calculate the jump address that is then passed to the program counter in the instruction fetch stage. Data memory access is divided into its own stage where the *load-store unit* (LSU) is used to perform the memory access. Similarly to the jump address, the address calculation for the data access is done in the execute stage. Finally, in the last stage, writeback, the result operand is written back to the register file.

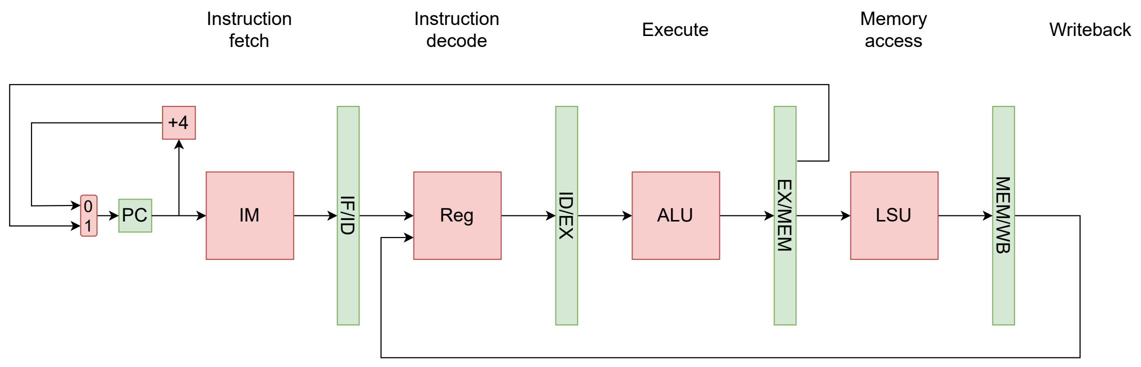

Figure 2.1. *Block view of a 5-stage RISC pipeline*

Even though pipelining increases the throughput of a processor, it does not come without its complications. During pipelined execution, there are situations in the processor pipeline when an instruction cannot be executed in the pipeline stage during a clock cycle without changing the program order. These situations are called *hazards*. Pipeline hazards can be divided into three groups: structural, data and control hazards. Structural hazards happen when multiple instructions in the pipeline would need the same resource. For example, the von Neumann architecture that has shared interface for data and instruction access would cause a structural hazard during a memory operation, as the memory access would have to be performed in the same clock cycle as the fetching of the next instruction. [2]

In the processor pipeline, data hazards are caused when an operation has a data dependency on the result of a previous operation that has not yet been written to the register file. An easy way to make sure valid operand value is assigned to an operation during a data hazard is to stall the processor pipeline until the result has been written to the register file. Pipeline stalls are also referred to as *bubbles*. This method induces a significant amount of stalls because data hazards are common during program execution. More sophisticated way to solve the data hazard issue is to forward the data to a previous stage in the processor pipeline straight from the *function unit* output port without routing the operand through the register file. This way, the pipeline can continue to operate and the correct operand is assigned to the execute stage. However, not all data hazards can be solved without stalls. [5] As in the pipeline in Figure 2.1, the memory access is divided into its own pipeline stage. If the next instruction has a data dependency on the load operation, the result operand of the load operation would not have yet arrived to the core when the result is needed as an input operand in the next instruction. In this scenario, the pipeline would have to be stalled even with bypass support from the memory access stage.

Control hazards are caused by control flow operations. The essential issue is that control flow operations can break the sequential flow of operations by causing a jump to a new instruction address, which causes a dependency on the next instructions in the pipeline. Some instruction set architectures use programmer visible delay slots to minimize the pipeline stalls and allow the programmer to insert useful instructions to cycles that would otherwise cause stalls in the pipeline. [2]

In the example pipeline, the hardware could deduct in the decoding phase that the instruction is a control flow operation. At this point, the instruction fetch stage is already fetching the next instruction. An absolute unconditional jump can be executed with no pipeline stalls if the jump address and control logic is passed directly from the decode stage to the instruction memory by bypassing the program counter register. This would, however, create a long *combinatorial path* in the design. If the jump is handled as in the example pipeline, it would cause three stalls.

Additional complexity is added by the use of conditional jumps that are also known as conditional *branches*. Conditional jumps are executed only if a condition that is set in the instruction is met. Usually, the branch condition uses a register file operand together with an arithmetical or a logical operation. [2] In the example pipeline presented in Figure 2.1 the ALU is used to calculate the branch condition which means that conditional branches would cause three stalls if the branch condition is controlled from the execute stage pipeline registers and passed to the program counter register. If the branch control bypasses the execute and the program counter registers, branches would only cause one stall cycle at the cost of longer combinatorial paths. The processing of branch conditions could also be moved to the decode stage, which would save one additional stall cycle.

Additional optimizations can be made to conditional branches because not-taken branches do not break the sequential flow of instructions. This way, during a conditional branch instruction, the pipeline can continue to operate. The issue is how to solve the hazard when a branch is taken. A way to deal with this is to flush the instructions in the pipeline if a branch is taken. Pipeline flushing can be implemented by extending pipeline stages with control logic that effectively transforms the invalid instructions into *no operations*. [2]

2.4 RISC-V

RISC-V is a relatively new open standard instruction set architecture that follows the RISC design philosophy. RISC-V instruction set is a modular structure, which is defined by the compulsory base integer ISA and optional extensions that can be added to support additional operations. The ISA also allows custom operations, which enables the design of *application-specific instruction-set processors* (ASIP) based on the RISC-V instruction set architecture. [6]

RISC-V specifies three variants to the base instruction set that differ in bitness: 32-bit, 64-bit and 128-bit. The 32-bit variant has two subsets: RV32I and RV32E. The RV32I is similar to the 64- and 128-bit variants because like the 64- and 128-bit variants, it has 32 general-purpose registers instead of 16 as in the RV32E subset that is targeted for small embedded applications. All of the variants reserve the bottom most register as a zero-register that has all bits hard coded to zero. The RV64I and the RV128I are built on top of the RV32I variant, but they have a wider *datapath*, address spaces and additional operations. [6] The decision for supporting multiple variants was driven by the popularity of 32-bit architectures in embedded systems and respectively 64-bit architectures' popularity in personal computers [7].

The different instruction formats of the RISC-V instruction set architecture are presented in Figure 2.2. RISC-V instructions use six different formats: R-, I-, S-. B-, U- and J-type. The formats have shared properties to simplify the decoding logic of the hardware. As seen in the figure, the bits indicating the operation code, function fields and register file indexes: rs1, rs2 and rd always exist in the same places in the instruction word between formats. The left-most bit of the immediate value is always the 31st bit in the instruction word to simplify the sign extension logic [6].

31 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]				rd	opcode	U-type
imm[20|10:1|11|19:12]				rd	opcode	J-type

Figure 2.2. *RISC-V 32-bit instruction formats [6]*

Because of the fixed placement of the register file indexes, the bits presenting immediate values are scattered in different places between instruction formats, which is why the immediate value must be shuffled based on the instruction format. This property is not unique to the RISC-V ISA as it was also used in the SPUR [8] architecture. Immediate values are presented in five different ways in instruction formats, but all the immediate values are sign-extended to the data width of the architecture. [6]

The control flow operations of the RISC-V ISA are presented in Table 2.1. As seen in the table, RISC-V base ISA has eight control flow operations, six of which are conditional. However, RISC-V does not have a separate operation for direct jumps. Instead, the *jump and link* (JAL) operation is used to implement direct jumps. JAL operation writes the return address as a result operand to the register file. If the operation is used as a direct jump instead of a function call, the result value can be stored to the zero-register to save space in the register file. The *add upper immediate to program counter* (AUIPC) operation works similarly, but in its case, the immediate value is not added to the program counter and instead the program counter is added to the immediate value and the result stored in the register file. [6]

The RISC-V control flow operations, apart from the *jump and link register* (JALR) operation, use program counter relative addressing, where the jump offset address is added to the program counter value. JALR operation can be used as a return statement because it uses an absolute address that comes from a register. [6] In addition to the program counter relative addressing, the control flow operations do not have visible delay slots because it is a microarchitectural pattern that does not offer major benefit for aggressively pipelined and superscalar implementations [7].

Instruction	Name	Description
beq	Branch ==	if(rs1 == rs2) PC += imm
bne	Branch !=	if(rs1 != rs2) PC += imm
blt	Branch <	if(rs1 < rs2) PC += imm
bge	Branch >=	if(rs1 >= rs2) PC += imm
bltu	Branch < (U)	if(rs1 < rs2) PC += imm
bgeu	Branch >= (U)	if(rs1 >= rs2) PC += imm
jal	Jump and Link	rd = PC + 4; PC += imm
jalr	Jump and Link Reg	rd = PC + 4; PC = imm + rs1

Table 2.1. *Control flow operations of the RISC-V base instruction set architecture [6]*

2.5 Instruction-level Parallelism

Instruction-level parallelism (ILP) is one form of parallelism and a way to increase the performance of a processor. In the example pipeline presented in Figure 2.1, the core would fetch one instruction per cycle and execute it in the pipeline. Pipelining can be seen as a form of instruction-level parallelism, as the execution of instructions overlap due to the pipeline stages. However, even with pipelining, the maximum *instructions per cycle* (IPC) is one. Multi-issue machines execute multiple instructions in parallel in the processor pipeline, which increases the maximum IPC.

To execute multiple operations in parallel, instruction-level parallel multi-issue machines need parallel function units in the processor pipeline. Figure 2.3 shows an example of a multi-issue pipeline. As seen in the figure, the LSU and ALU are placed in parallel in the execute stage. This allows the pipeline to execute a memory and an arithmetic operation concurrently.

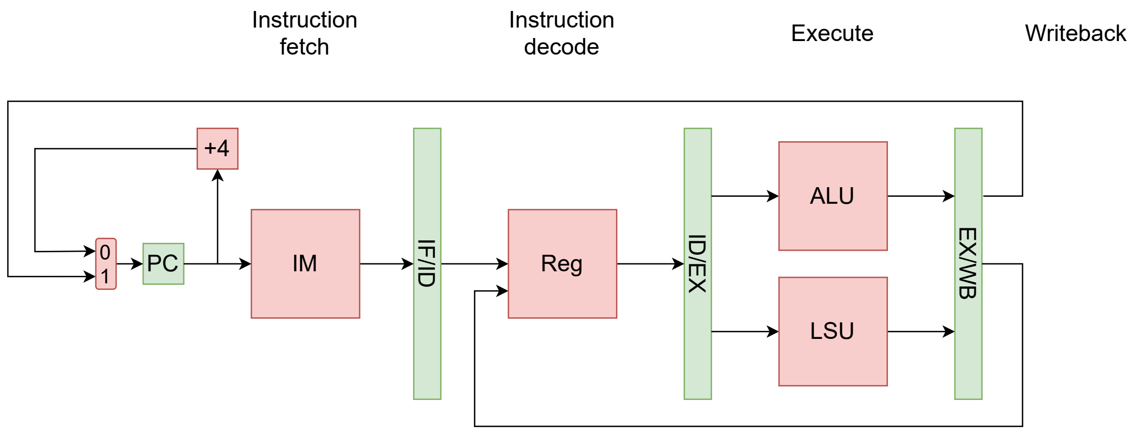

Figure 2.3. *Block view of a multi-issue pipeline*

Instruction scheduling is the process of deciding in which sequence the instructions are executed. Processors can be divided into *statically scheduled* and *dynamically scheduled* processors. In statically scheduled processors, the compiler expresses the sequence in which the instructions are executed. In dynamically scheduled processors, the hardware sequences the instructions during run time. The differences between these approaches are important when exploiting ILP. This section explores the ways to implement instruction-level parallelism in processors, focusing on superscalar and *very long instruction word* (VLIW) processors and inspects *transport triggered architecture* (TTA) as a variation of a VLIW processor.

2.5.1 Very Long Instruction Word Processors

VLIW processors are statically scheduled multi-issue processors where the instruction level parallelism is explicitly stated in the instruction word. In VLIW architectures, one long instruction word packs multiple operations that are then executed in parallel in the processor core. An enormous benefit of VLIW processors is the static scheduling that is determined by the compiler when the operations are packed into the instruction. This allows to explicitly exploit ILP without complex hardware that does the scheduling during run time. [9]

A big drawback of VLIW processor is code density, as the packets that cannot be fully utilized with operations are filled with no operations. The no operations can fill a sizeable portion of the instruction code, which is why some VLIW architectures use templates with differing amount of operations. [9] To support differing amounts of operations in instructions, the architecture can utilize variable-length instructions, which complicates the fetching and decoding of instructions as in some CISC implementations.

Transport Triggered Architectures

Transport triggered architecture follows the VLIW principle, where instructions are statically scheduled. TTA, however, is not based on the *operation triggered* model that is used in RISC architectures, instead the programming model is based on the transportation of operands. Operation triggered architectures are programmed by specifying an operation which results in implicit data-transports between the register file and function units. Transport triggered architectures have a lower level programming interface, where the datapath is exposed to the programmer. In the TTA programming model, the programmer states explicit operand moves between the function units and registers, which causes an execution of operations as a side-effect. [10]

Because of the exposed datapath, the programmer is aware of the interconnection network. In operation triggered architectures, the datapath is not visible in the programming interface even though it influences performance when the missing bypass connections cause stalls during data hazards. The connectivity of the interconnect plays a crucial role in programming the TTA processor as it dictates which moves are supported by the architecture. The customization of the interconnect network connectivity is an important feature as it contributes to the design area and possibly the critical path. Reducing the datapath connectivity is especially important for wide-issue machines, as the multiple combinations of parallel function units cause many combinations of bypass paths. [11]

Figure 2.4 shows an example of the modular structure of a transport triggered architecture. The structure is divided into multiple different building blocks: function units, register files, interconnect busses and sockets. The example design has three function units: an ALU, a LSU and a *control unit* (CU). The register file is treated similarly as a function unit, which allows the programmer to directly transfer operands between the register file and function units via the interconnection network [11]. The sockets connect the function unit ports to the interconnect busses. In the example architecture, the design has six interconnect busses and three parallel functions units. The socket connections are configured so that the busses can be used in parallel for transportation of operands to different functions unit which enables the exploitation of ILP. Due to the modular design philosophy of transport triggered architectures, the architecture can be easily scaled by adding more function units and interconnect busses to the design.

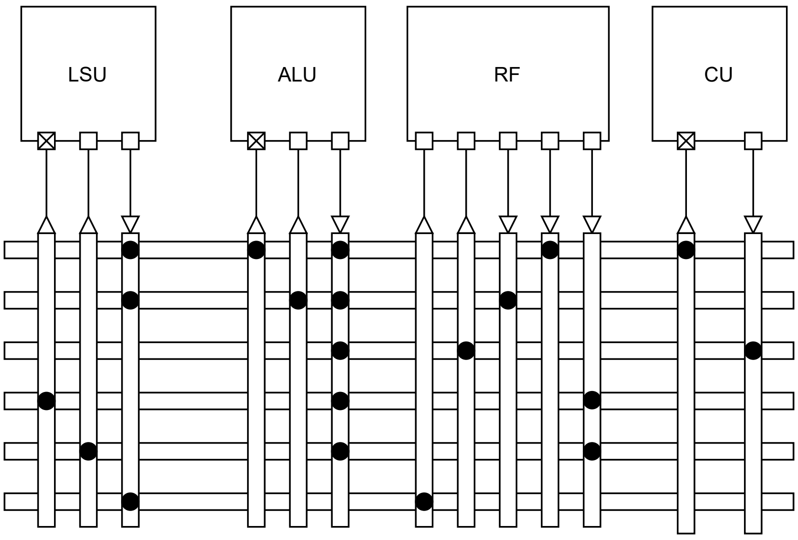

Figure 2.4. *Example of a transport triggered architecture*

The function units in the architecture implement one or more operations. The operations can be internally pipelined inside the function unit because they are implemented separately from the interconnect network. The operation latency is visible in the programming interface and the programmer must be sure that the result operand is moved from the output port after the operation has been executed in the operation pipeline and arrived to the output register. [11]

TTA instructions consist of moves that transport operands to and from the ports of a function unit. An operation is executed as a side-effect when an operand and the operation code is transported to the triggering port that are marked with crosses in the example architecture. Due to this operation model a separate move is needed for each input operand that can be transported in parallel if the interconnect connectivity allows it. The result operand is moved on a later cycle after the operation has been executed in the function unit. An addition operation that would be described in RISC-V as a single assembly instruction:

```
add r3 r1 r2
```

Can result in three separate instructions for a TTA:

```
Cycle 0: RF.r1 -> ALU.in
Cycle 1: RF.r2 -> ALU.t.add
Cycle 2: ALU.out -> RF.r3
```

If the interconnect has enough busses and the required connectivity, the two input operands can be transported during the same clock cycle, which reduces the amount of instructions to two:

```
Cycle 0: RF.r1 -> ALU.in    RF.r2 -> ALU.t.add
Cycle 1: ALU.out -> RF.r3   ...
```

The second bus could not be used to transport any operands in cycle one, which is why it was assigned with a no operation that is described with three dots in the code.

In operation triggered architectures bypasses are dynamic and done automatically by the forwarding logic when a data hazard is encountered. Due to the programming model of TTAs, the bypasses are programmable and therefore invoked by software. In a simple addition that would cause a data hazard in an operation triggered architecture, the hazard is hidden from the programmer, and the hardware can either stall the pipeline until the result operand of the previous instruction is written into the register file or forward the result operand:

```
add r3 r1 r2
add r4 r1 r3
```

On a TTA, the bypasses are generated by the programmer:

```
Cycle 0: RF.r1 -> ALU.in    RF.r2 -> ALU.t.add
Cycle 1: ALU.out -> ALU.t.add    ...
Cycle 2: ALU.out -> RF.r4   ...
```

In cycle one, the result of the previous operation is not transported to the register file at all, as it is not used by any other future instruction. This TTA-specific optimization is called dead result elimination. The above code also uses a technique called operand sharing. As the r1 input operand was already transported into the function unit port, it was not required to move it again on a later cycle. As seen in the assembly examples, the lower level programming model offers more scheduling freedom and optimizations to the compiler compared to traditional operation triggered architectures.

In the example architecture, the core has two register file write ports and three read ports. Multiple issue processors are known to have multiple register file ports to transport the operands to the parallel function units. Operation triggered VLIWs with N function units would need 3N register file ports if each function unit uses the maximum of two input and one output value. However, transport triggered architectures are less dependent on the amount of register file ports because operands are not required to be routed through the register file like in operation triggered architectures. TTAs are less dependent on accessing the register file due to the additional scheduling freedom and optimizations enabled by the lower level programming interface, which reduces the amount of register file operands in the program code. In addition to the reduction of register file size and port amount, TTA-specific optimizations increases energy efficiency as accesses to the register file are reduced. [11]

An example of a transport triggered architecture's instruction format is presented in Figure 2.5. The instruction word is divided into different move slots that present flow of data in the core's interconnect busses. The move slots have separate source and destination fields that describe the source and the destination sockets. The destination field also specifies the operation code of the targeted function unit if it is connected to a triggering port. If a move slot cannot be used for the transportation of an operand in an instruction, the move slot is assigned to a no operation.

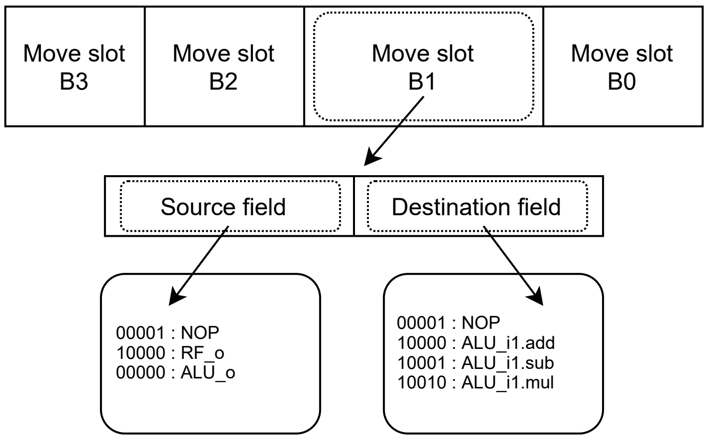

Figure 2.5. *Example of a transport triggered architecture's instruction format*

2.5.2 Superscalar Processors

Superscalars, also known as dynamically scheduled multi-issue processors, take a different approach to instruction-level parallelism compared to VLIW processors. In superscalar processors, the ILP is not explicitly stated by the compiler. Instead, the core receives the same instructions as an equivalent single-issue processor of the same ISA. Superscalar processors exploit ILP by fetching multiple instructions to an instruction queue during the same clock cycle and dynamically scheduling them in hardware so that multiple instructions are executed in parallel when possible. [9]

An enormous benefit of superscalar processors is that as the multi-issue capability is purely an implementation detail that can be hidden from the programmer, the design is compatible with the binaries with different dynamic multi-issue or single-issue implementations of the same ISA. The dynamic scheduling of operations, however, results in more complex hardware implementations, which is a big drawback of superscalar processors. [9] The more complex hardware implementation of superscalar processors is problematic especially when targeting embedded devices and optimizing for low power consumption.

3. PROCESSOR CUSTOMIZATION

Processor customization is a way to optimize processor implementations and architectures towards the desired use case. While customization can yield better results in terms of performance, area and energy efficiency, it is a time-consuming and error-prone task. This chapter explores ways of processor customization and available RISC-V generators as well as the OpenASIP toolset as an example of processor customization tools.

3.1 Application-specific Instruction-set Processors

The term application-specific instruction-set processor is not strongly defined. However, in literature it is usually used as a term for a processor whose instruction set is tailored for a specific application domain [12] [13] [14]. Compared to *general-purpose processors* (GPP) whose instruction set is designed to achieve the maximum performance and flexibility in general-purpose computing, ASIPs can achieve better performance and energy-efficiency in the target domain while possibly losing some of the flexibility that comes with general-purpose processors. The key design benefit of ASIPs is the ability to tailor the instruction set in a way where instructions that are not beneficial in the target domain are removed and respectively custom instructions that accelerate the target applications can be added. This way, the area and performance are strongly optimized for the application domain.

Overall, the flexibility, performance and power consumption of ASIPs falls in between GPPs and non-programmable fixed function accelerators. ASIPs benefit from the flexibility gained from programmability even though it comes with an overhead in area, performance and power consumption. Implementing ASIPs is also less risky and offers a shorter time-to-market than fixed function *application-specific integrated circuits* (ASIC) as debugging software is cheaper than post fabrication debugging of hardware. In addition, ASIPs can be theoretically produced in higher volume compared to fixed function accelerators because related applications in the same domain can use the programmable hardware for acceleration. [15]

The tailoring of the instruction set in ASIPs does not come without a cost because the tailored instruction set must be supported by the compiler and the instruction set simulator so that the processor can be used efficiently. This adds motivation for ASIP design environments that can automatically generate the software development kits from a higher-level description of the processor.

3.2 Architecture Description Languages

The term *architecture description language* (ADL) has been used for designing of both hardware and software architectures. In hardware architectures, ADLs are used to describe hardware components, their connections as well as the behaviour. It is used in a similar manner for software architectures where ADLs describe the behavioural specifications and interactions of software components. There are multiple terms for ADLs that target processor design, such as processor and machine description language. Even though the concept of ADLs is not strongly defined, they are used for describing systems on a higher level where architectural information is presented rather than the implementation itself, as in hardware description languages. [16]

Using ADLs in processor design is good for design space exploration, as the designer can explore the processor on an architectural level without modifying the microarchitectural details. In addition to the hardware customization and generation, ADLs make the automatic generation of testing environments and software toolkits easier for customized processors as all the architectural information is known in the architecture description. This is especially important when developing retargetable compilers to add compiler support for ASIPs. [16]

One way of classifying ADLs is their objective. From this perspective, ADLs can be divided into compilation-, simulation-, synthesis- and validation-oriented ADLs. The main purpose of compilation-oriented ADLs is to enable automatic generation of retargetable compilers where the ADL is used to provide the compiler information about the architecture as input. Simulation-oriented ADLs are used for simulating customized processors. Simulation can be divided into multiple abstractions where the higher level abstractions produce functional simulation and the lower level abstractions clock cycle accurate information. The synthesis-oriented ADLs are used for hardware generation and validation-oriented for functional verification of processors. Many ADLs, however, have a mix of these objectives. [16]

LISA is an example of a mixed-level ADL that describes the behaviour, structure and the interfaces of a processor architecture. The LISA model is divided into two main parts. The first part describes the resources of the processor architecture, while the second part stores information about the instruction set, behaviour, expression and timing in the form of operations. The resource entries consist of multiple subsets that include registers, pipelines and memories that can be parameterized with different values. The operation descriptions can be further divided into multiple sections: coding, syntax, semantics, behaviour and activation. The coding section is used to describe the binary image of an instruction word, the syntax section for describing the assembly syntax and the semantics section for expressing the abstracted behaviour of an instruction. The behaviour and expression sections describe state transitions, and the activation section is for describing the activation of instructions in the pipeline. Effectively, the processor model is divided into multiple submodels that describe different parts and abstraction levels of the processor. [17]

3.3 Processor Generation and Customization in OpenASIP

OpenASIP [18] or TCE is an open source TTA-based application specific instruction-set processor toolset that allows users to generate and program customized ASIPs. OpenASIP allows heavy customization of both the architecture and implementation of the processor.

As seen in Figure 3.1, the processor customization is divided into multiple different tools and files in OpenASIP. The most visible tool to the user is the Processor Designer that allows to customize the architecture of the processor. Processor Designer provides a graphical user interface for modifying the XML-based *architecture definition file* (ADF) that has all the information about the programming interface of the processor and is used for both compilation and simulation in addition to the hardware generation. ADF stores information about the interconnect network, function units, their operations and latencies, memory sizes and register files. [19]

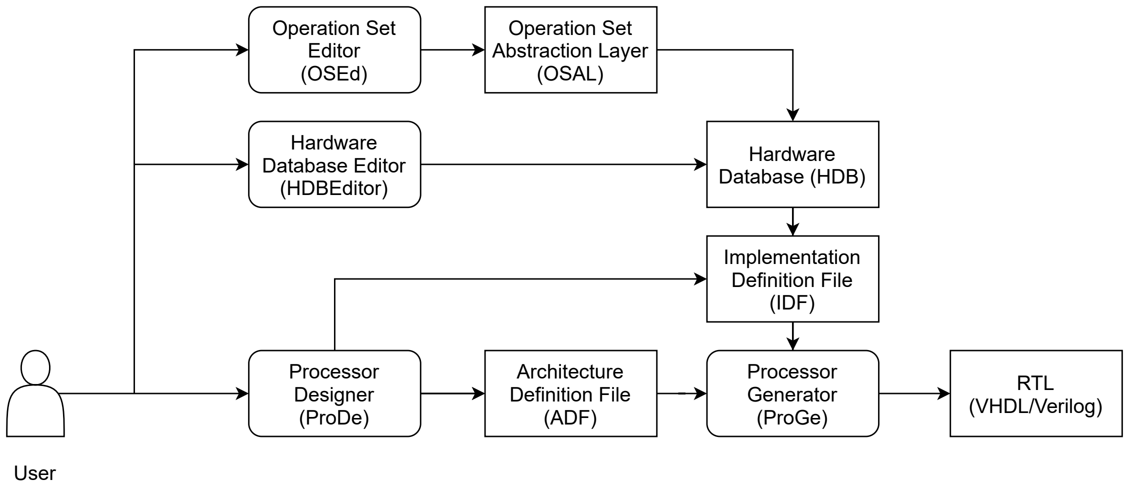
Operation Set Editor (OSEd)
Operation Set Abstraction Layer (OSAL)
Hardware Database Editor (HDBEditor)
Hardware Database (HDB)
Implementation Definition File (IDF)
Processor Designer (ProDe)
Architecture Definition File (ADF)
Processor Generator (ProGe)
RTL (VHDL/Verilog)
User

Figure 3.1. *Overview of processor generation and customization in OpenASIP*

In addition to the architectural modification, OpenASIP has separate tools for modifying operation set libraries and the hardware databases. Operations can be added to the operation libraries with the Operation Set Editor that is operated via a graphical user interface. In OpenASIP, the operations are strongly separated from their hardware descriptions so that not even the operation latency is described in the *operation set abstraction layer*

```
<operation>
  <name>MAC</name>
  <description>Multiply and accumulate (signed integer).</description>
  <inputs>3</inputs>
  <outputs>1</outputs>
  <in element-count="1" element-width="32" id="1" type="SIntWord"/>
  <in element-count="1" element-width="32" id="2" type="SIntWord">
  <in element-count="1" element-width="32" id="3" type="SIntWord">
  <out element-count="1" element-width="32" id="4" type="SIntWord"/>
  <trigger-semantics>
    SimValue mul_result;
    EXEC_OPERATION(mul, IO(2), IO(3), mul_result);
    EXEC_OPERATION(add, mul_result, IO(1), IO(4));
  </trigger-semantics>
</operation>
```

Figure 3.2. *Multiply and accumulate operation entry*

(OSAL) and therefore only the semantics of the operation are described in the operation description. The hardware implementations for function units and operations are described separately in the hardware databases that can be modified with the Hardware Database Editor. [19]

OSAL stores the semantics and interfaces of operations, which gives it a key role when adding custom operations. The static properties of operations are added to an XML-based .opp file that describes the operation name and interfaces. The operation semantics can be described as a *directed acyclic graph* (DAG) in the .opp file if the operation can be constructed by combining different pre-defined OSAL operations. Otherwise, the operation behaviour model must be described in a separate .cc file that is used to describe the operation behaviour. [19] An example of a multiply and accumulate is presented in Figure 3.2. The entry states that the operation takes three 32-bit input values and emits one 32-bit output value. Additionally, the semantics of the operation are described under trigger-semantics where the mul and add operations are used to describe the operation as a DAG.

The implementation of the processor is defined in the *implementation definition file* (IDF). IDF stores all the information about the implementation that is not relevant in the programming interface, such as hardware implementations of the function units and register files. Like the ADF, the IDF is an XML-based file that can be either modified manually or in the Processor Designer tool. [19]

In the last step, the command-line tool Processor Generator is used together with the ADF and IDF as main input to produce the *register transfer level* (RTL) description of the processor. The OpenASIP hardware generation can produce the hardware descriptions both in VHDL and Verilog. [19]

3.4 RISC-V Generators

The open-standard nature and rising popularity of RISC-V has created motivation for customizable implementations and core generators. This section explores available commercial and open source RISC-V generators and compares their features.

As seen in Table 3.1 there are already many tools that allow the generation of customizable RISC-V implementations. The tools have many common features, even though some of them are more focused on full *system-on-chip* (SoC) implementations.

Codasip Studio [20] is a commercial tool for generating customizable RISC-V cores and software development kits for the generated hardware. Codasip uses a high-level description language CodAL that can be used to describe different kinds of instruction-set architectures in addition to RISC-V. [21] Even though Codasip Studio is a commercial tool, it has also been used in academic work to design an application-specific instruction-set processor for 5G data link layer processing [22] as well as to implement an instruction set extension for the secure hash algorithm for the MIPS instruction set architecture [23]. Codasip Studio has a strong support for custom operations and is able to automatically generate the hardware for the custom operations as well as integrate them into the LLVM-based compiler toolchain without the need for intrinsics in the source code.

SiFive Core Designer [24] is another commercial tool for generating customized RISC-V implementations from multiple different core templates with a vast amount of customization points. The templates can be modified to include multiple RISC-V cores and configure many parts of the internal microarchitecture such as branch predictors, caches and debuggers.

Andes [25] RISC-V core customization works in a similar way as SiFive's, where the processor is modified from a processor template. However, the templates are more fixed and do not allow users to heavily customize the internal implementation as in SiFive's Core Designer. The user can add custom operations to the processor templates with instruction development tools that configure the compiler toolchain and RTL.

Synopsys ASIP Designer [26] is also a commercial tool that allows heavy customization. ASIP Designer is based on the nML architecture description language and contains many other processor templates besides RISC-V. ASIP Designer ships with a retargetable compiler and a simulator that are configured based on the architecture description. [27] ASIP Designer can be extended with MP Designer [28] to add support for multicore designs.

WARP-V [29] is an open source tool that allows the user to generate customized RISC-V cores. The tool supports only generating the core logic and does not support platform components, such as caches and memory management units. The generator utilizes TL-Verilog to describe the core architecture and even has support for generating multicore designs. The WARP-V does not support custom operations and does not offer compiler support like SiFive Core Designer, Andes, ASIP Designer and Codasip Studio.

Rocket Chip Generator [30] is another open source tool developed by the University of Berkeley. It utilizes the Chisel hardware construction language to combine a library of generators for cores, caches and interconnects into a SoC implementation. Rocket Chip Generator has been used to produce functional ASIC implementations that are capable of booting Linux. The tool is divided into multiple different generators that handle different components. The Core Generator is used to instantiate and customize RISC-V cores. It offers customization for function unit pipelines, branch predictors and floating point units. The toolset has multiple different core generators that use different base implementations: Rocket core that is a scalar core with a 5-stage in-order pipeline, BOOM, that is an out of order superscalar core and Z-scale that is a smaller 3-stage core.

Another interesting implementation is VexRiscv [31] that is a SpinalHDL [32] based RISC-V implementation. SpinalHDL is a scala library that enables to describe hardware implementations. VexRiscv describes the different parts of the RISC-V core as plugins, which allows heavy customization. However, it is not exactly a generator even though the RTL is generated from the SpinalHDL description and therefore requires the user to manually modify the description to customize it.

Overall, there are not many open source tools for generating customized RISC-V implementations. Many of the tools are commercial and neither freely available nor extensively documented. The missing support for custom operations was also observed in open source tools.

	ASIP Designer	Codasip	SiFive	Andes	WARP-V	Rocket
Custom operations	x	x	x	x		
Multicore	x		x	x	x	x
Configurable pipelining	x	x	x	x	x	x
Branch prediction	x	x	x	x	x	x
Caches	x	x	x	x		x
Open source					x	x

Table 3.1. *Properties of available RISC-V generators*

4. HARDWARE GENERATION AND IMPLEMENTATION

To enable generation of customized RISC-V implementations, this work extends the open source tool OpenASIP. In the implementation of the RISC-V generation, a TTA core is used as the internal microarchitecture and a RISC-V front end is generated to implement part of the control and decoding logic. This enables reuse of the customization points that are available for TTA cores in OpenASIP. Transport triggered architectures are a suitable candidate for describing more high-level architectures because of their exposed datapath that allows the programmer to directly move data between different function units and the register file via the core's internal interconnect.

The main benefit of extending OpenASIP and using a TTA core as base implementation is the easy design time exploration. In practice, the front end acts as a design time microprogramming layer in the hardware that can be optimized during the synthesis phase. The method follows a similar microprogramming design philosophy that is used in CISC implementations to design control logic. In this work, however, instead of RISC-like micro-operations, the internal micro-operations are TTA moves. Using the microprogramming to design control logic does not offer any runtime benefits in this work as the microcode is not programmable and is merely used to design the control logic for RISC-V implementations. This chapter explains how the microcode component was implemented and how its generation was integrated into the OpenASIP toolset.

4.1 Processor Pipeline

A high level description of the design pipeline with RISC-V microcode support is described in Figure 4.1. The pipeline is divided into four pipeline stages where both the translation and decoding of the micro-operation are done in the same combinatorial path. Because of the programming model of TTA cores, they don't have the implicit writeback functionality in hardware as in operation triggered architectures. However, due to the added microcode component, the writeback stage is implicitly formed because the microcode hardware guarantees that the result operand is always written to the register file.

A microarchitectural difference compared to the classic RISC pipeline is caused by the register file placement in the core pipeline. In TTA cores, the register file is treated similarly as function units, which enables the programmer to directly move data from and to the register file ports via the core's interconnect. In the described pipeline configuration due to the decode registers, the register file read is done in a different stage than the decoding of the instruction. In classic RISC implementations, the register file is usually read in the decode stage. Because of the operation model of TTAs, the register file cannot be moved to the same stage as the instruction decode without removing the decode stage registers, as then the register file would not be accessible via the interconnect. If the decode registers were removed, the pipeline would resemble a moderately pipelined RISC implementation at the cost of a longer combinatorial path.

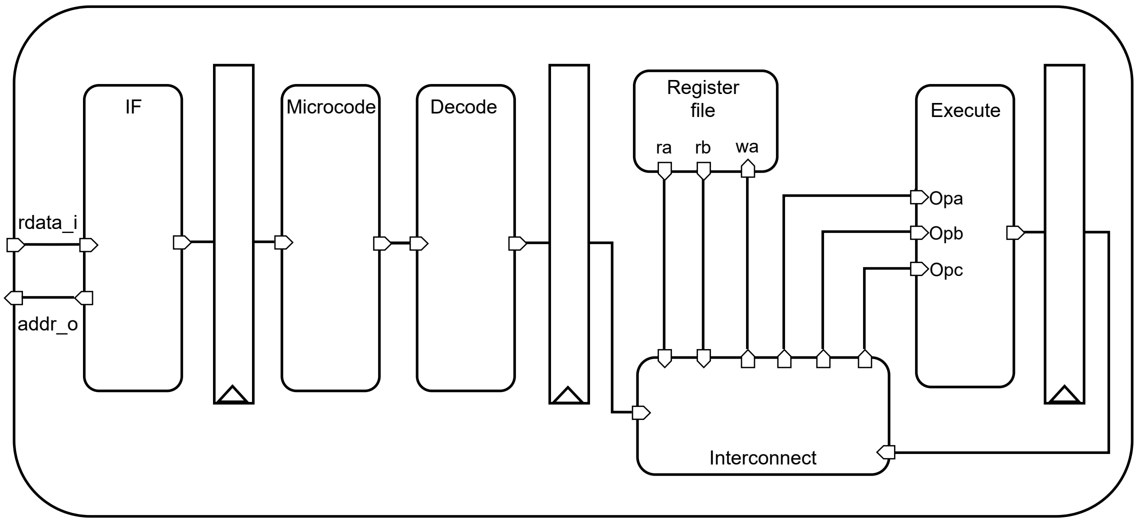

Figure 4.1. *High-level description of the design pipeline*

The RISC-V instruction set architecture specifies the instruction width, which is why the width of the fetched instruction blocks is fixed to 32 bits. The micro-operations that are emitted from the microcode hardware can be viewed as TTA moves, but here they are used for implementing the control logic of an operation triggered architecture. Essentially, they are a step of forming the control signals to the core pipeline that are emitted from the decode unit.

In this work, the microcode hardware must implement the hardware features that are not found in transport triggered architectures which include dynamic features such as data hazard detection, data forwarding and handling of control flow operations in a way that is specified in the instruction set architecture. During control flow operations, the microcode hardware must assure that the control hazards are handled in a way that the program order is preserved. In the RISC-V ISA, control flow operations do not use delay slots and therefore the microcode hardware must make sure that the pipeline is stalled if a control hazard arises.

4.2 Microcode Implementation

The microcode component must provide a programming interface for the target instruction set architecture. Because of the vast differences in the operation models of operation and transport triggered architectures, the instruction binaries cannot be statically translated and instead the multiple micro-operations must be sequenced separately to achieve the wanted operation-triggered behaviour. Some features that are software programmable in TTA cores are handled by hardware in operation triggered cores, for example, data forwarding.

Figure 4.2 shows a block diagram of the internal structure of the microcode unit. Some details and interfaces that are not directly related to the translation and sequencing of the micro-operations are hidden from the block diagram to simplify it. As seen in the figure, the microcode component has multiple different subcomponents that are explored in more detail in later sections. The most important components are the controller that is responsible for handling the control flow operations, the micro-operation sequencer that schedules the micro-operations and lookup tables that contain the microcode and operation latencies. Additionally the decoding of the formats and data hazard detection is done inside the microcode component.

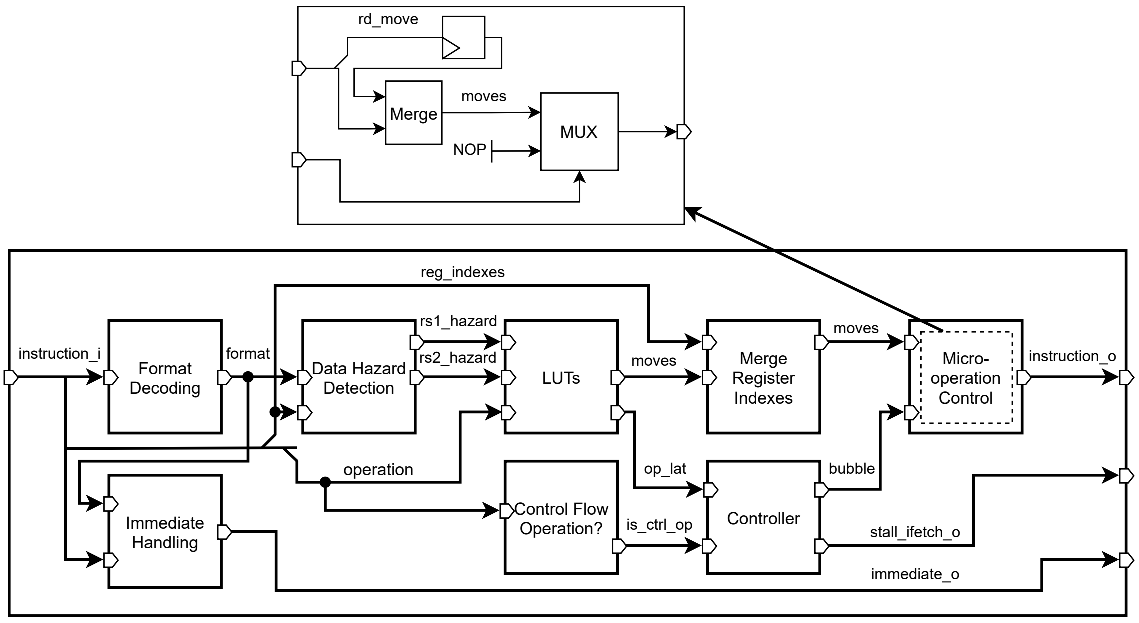

Figure 4.2. *Block diagram of the microcode hardware*

A similar method has been used to interpret x86 instructions on an ARM core, where the ARM instructions can be seen as micro-operations. The method uses translation tables that transform the x86 into one, two or three ARM instructions that are then executed by using a micro-operation sequencer. The interpretation hardware effectively serves as an x86 front end to the ARM core. The implementation includes two operation modes where both ARM and x86 instructions can be run on the same hardware. [33] In this work, however, the architecture is fixed to the RISC-V ISA without allowing multiple operation modes to focus on the generation of standard RISC-V implementations.

4.2.1 Instruction Translation

The most important component of the microcode hardware is a lookup table that maps instructions between one another. The instruction word that is read from the lookup table is later split into multiple micro-operations that are scheduled separately. Adding an entry in the lookup table for every possible combination is not feasible as immediate values and register file indexes would cause too many combinations and, therefore, make the lookup table impractical for hardware generation and synthesis. When both instruction set architectures support same ranges for register file indexes and immediate values, they can be directly mapped between instruction words without routing them through a lookup table. This reduces the lookup table size and now only combinations of instruction formats and operations need an entry in the lookup table.

When the immediate values and register file indexes are removed from the RISC-V instruction word, only the operation code and function fields are left. Respectively, TTA moves consist only of the operation sources, destinations and the operation code on the triggering bus when the register file indexes and immediate values are excluded. In this case, the TTA moves can be mapped solely from the RISC-V operation code and function fields as they identify these properties.

For the translation to work correctly, the microcode unit must be aware of how the different operands are mapped to the transport busses. This requirement is due to the direct mapping of the register indexes as well as the splitting of the translated instruction into multiple micro-operations.

When a new instruction arrives to the translation stage, the immediate values and register indexes are sliced from the RISC-V instruction word. The register indexes are mapped directly to the translated TTA instruction. Handling of register indexes in hardware is easy because in RISC-V's case the register file indexes exist in the same places in the instruction word between formats as observed in Figure 2.2. Handling of the immediate values is more complex as each format has a different way of expressing immediate values, which is why the immediate bits must be shuffled and shifted based on the instruction format. The immediate values are not inserted into the translated instruction word, instead, they are passed directly to the decoder output. This way handling of the immediate value is independent of the supported immediate width by the internal instruction format of the microarchitecture as the immediate value is sign extended to 32 bits and forwarded to the decode output by the microcode unit.

4.2.2 Micro-operation Sequencing

The differences in the programming model between operation and transport triggered architectures cause additional complexity because instructions cannot be translated directly between one another. During the scheduling of the micro-operations, the hardware must assure that the result is transported into the register file during the correct clock cycle. In practice, this means that the translated micro-operations must be scheduled in two sequences where the micro-operation that moves the result operand back to the register file is scheduled on a later cycle. Implementing such a control structure is simple for hardware implementations where all operations have an operation latency of one clock cycle. In this case, the result move can be forwarded into a register that delays the result move from being passed to the decoder by one cycle, ensuring that the operation has been executed in the execute stage when the result move is performed.

In Figure 4.2 the sequencing of micro-operations is shown in more detail. The result operand move is sliced from the translated instruction and assigned to a register, the input operand moves are passed directly to the decoder. The controller inside the microcode unit can insert a bubble that bypasses the sequencer output in order to bubble the pipeline.

However, it is common that the operation latency differs between operations. Splitting complex operations into multiple cycles is one way of shortening the critical path in hardware implementations. This is commonly used for division and multiply operations that would otherwise cause a long combinatorial path in the design with the possible cost of stalls when the operation is executed in the operation pipeline. Multi-cycle operations cause additional complexity to the micro-operation sequencing, as the result move cannot be statically delayed. Instead, during the translation of the incoming instruction, an additional lookup table must be read to find out the operation latency for the incoming operation and bubble the pipeline until the result move can be executed.

Additional steps must be taken to add support for the RISC-V load upper immediate operation that loads 20 bits from an immediate value to the destination register upper bits by using the U-format and replaces the lower bits with zeroes. [6]. In transport triggered architectures, this operation could be described as a simple move between a short immediate and the register file. However, it is not an optimal solution to describe the operation in such a way by the microcode hardware because the immediate move would have to be mapped to the result operand bus. If the bus that is used for the result operand moves has support for short immediate values, it could be used to delay the result move like with other operations. A more general solution is to mimic the way small immediate values are loaded into the register file in RISC-V applications. The programmer can use the add immediate operation and mark the other input operand as the zero-register which loads the original immediate target value to the register file by routing it through the ALU.

This same method can be used internally to solve the load upper immediate scheduling problem without adding extra register file write ports, short immediate support for the result operand bus or complicating the micro-operation sequencing. This design choice does not come without its drawbacks, as now loading the upper immediate value causes extra switching activity in the ALU, potentially resulting in a higher energy consumption.

4.2.3 Control Flow Operations

Control flow operations are the most complex types to sequence because they propagate into the core's control logic. To minimize the amount of required stalls, the control flow operations should be handled as a special case. This is easy to implement for operations whose input operands have no dependency on the register file. This way, they can be assigned directly to the control unit and added to the program counter value without routing them via the other stages of the processor pipeline.

Both the JAL and the AUIPC operation can be directly routed to the control unit to minimize control hazard related stalls. However, the result move must still be scheduled to ensure that the result operand is stored in the register file after the operation has been executed.

JALR, as well as branch operations, are more complex to optimize because their input operands depend on register file values. As seen in 4.1, in TTA designs, the register file is treated similarly as a function unit, which means that register file dependent operations must be routed through the interconnect. A way to optimize this issue is to predict that the branch is not taken and keep the pipeline running. In case the branch was mispredicted, the following instructions of the branch instruction would be flushed out of the pipeline. A similar optimization cannot be made for the JALR operation because it utilizes an unconditional jump.

In this work, when the microcode hardware encounters a branch or a JALR instruction, the instruction fetch unit is stalled and the micro-operation inserted into the core pipeline. Because of the decode registers, the core must bubble the pipeline for one clock cycle until the operation has been executed in the control unit. After this, the previous stages must be filled with valid instructions, which takes one to two cycles depending on whether the instruction register is enabled in the instruction fetch unit. With the described configuration, the core suffers a penalty of N-1 cycles, where N is the amount of pipeline stages. The amount of pipeline stalls could be minimized by reducing the amount of pipeline stages, but this is not an optimal solution for general-purpose performance as it would cause long combinatorial paths to the design, which reduces the maximum clock frequency. Additionally, the program counter register could be bypassed during control flow operations, but this could form a long combinatorial path when memories are connected to the core.

4.2.4 Data Hazards and Forwarding

TTAs' specialty are the software programmable bypasses where the programmer can assign a move from a function unit output port to a function unit input port without routing the data through the register file. This is possible when the interconnect has the required connectivity. A similar method can be used to perform data forwarding if the microcode unit detects a data hazard. When a data hazard is detected, the move from the register file to the function unit input port on the data hazard bus is not assigned. Instead, the operand is routed from the function unit output port by utilizing the bypass connectivity.

A core with full bypass support to every register operand is presented in Figure 4.3. As seen in the figure, the register file output ports are connected to the first and the second bus in the architecture. Respectively, all function unit output ports are connected to the first and the second bus, as well as the third bus that is connected to the register file input port. This way, the executed results can be forwarded straight from the function unit output port when a data hazard is encountered. The core presented in Figure 4.4 has no bypass support as the function unit output ports are only connected to a bus which only input connection is to the register file.

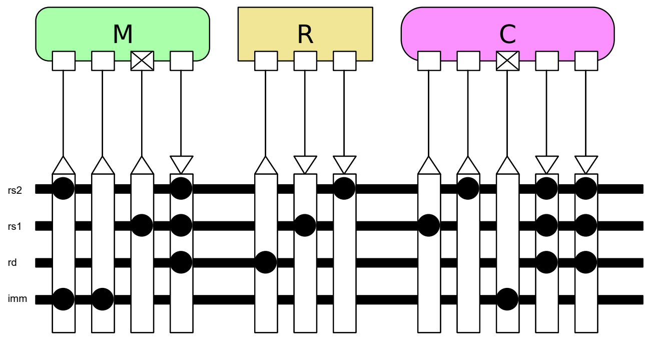

Figure 4.3. *Architectural view of a core with full bypass support*

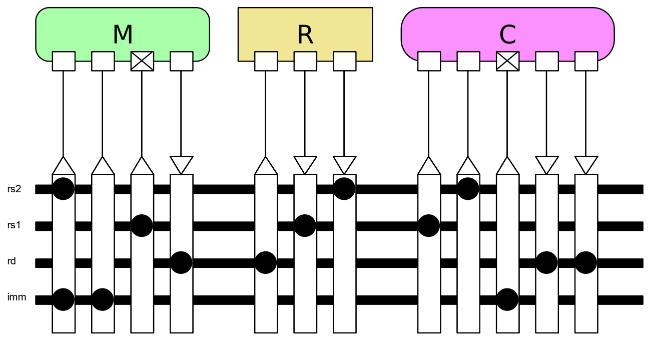

Figure 4.4. *Architectural view of a core with no bypass support*

To make the dynamic forwarding possible, the microcode unit needs additional lookup tables. In the first lookup table, each operation is assigned a target tag which identifies which output port the operation stores the result operand after it is executed. This way, the microcode hardware can use a register to store the output port of the previous operation. In addition, a lookup table is needed for each register input operand: rs1 and rs2. The operand lookup tables have an entry for each combination of operand output and input ports. When encountering a data hazard, the move from the register file is discarded on the hazard bus and bits for that move are fetched from the bypass lookup table. It is important that the result move to the register file is still simultaneously performed alongside with the forwarding move.

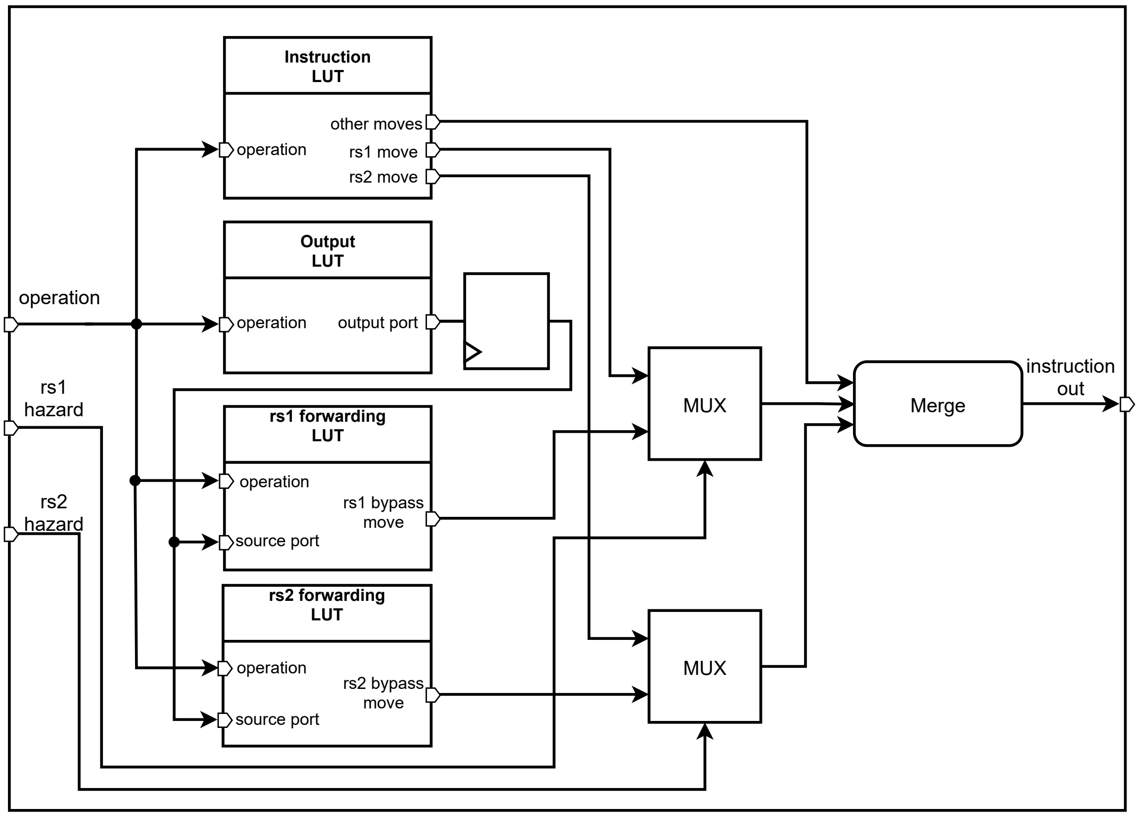

Figure 4.5. *Instruction translation with data forwarding support*

The dynamic bypass logic is described in Figure 4.5. By default, the translated instruction is fetched from the instruction lookup table. During each clock cycle, the output port for the operation is read from a lookup table and assigned to a register. When a data hazard is encountered, the output port of the previous operation is passed to the operand bypass lookup table as the source port for the bypass move. The forwarding moves are passed through a multiplexer together with the default register file move, which enables the hardware to discard the move from the register file on the data hazard bus and use the bypass move instead. The data hazard is detected in a separate component in the microcode hierarchy, as presented in Figure 4.2. The data hazard detection unit is passed the register indexes and the current instruction format, which are used to deduct whether the current operation will result in a data hazard. The data hazard unit has internal registers that store the format and result index of the previous operation.

4.3 Microarchitectural Patterns

ADF allows users to heavily customize the architecture of the generated ASIP cores. However, to fully meet the requirements of the RISC-V instruction set architecture, the TTA microarchitecture that is used in the implementation must meet certain requirements.

The most obvious requirement is that the underlying TTA microarchitecture must have all the required operations to implement the RISC-V base instruction set architecture. The microcode unit could implement the missing operations as multiple entries of microcode that would start a sequence of instructions to implement the missing operation with several operations. For instance, this could be done for missing logical operations. Implementing operations as microcode would complicate the translation hardware and decrease performance, which is why it is not implemented in this work.

RISC-V instruction formats dictate how many and which operands the bound operation must take as input and whether the operation emits an output value. This causes a requirement for the amount of busses and their configuration. To achieve the best performance and minimize the micro-operation sequencing logic, the core should be able to transport all the operands during each clock cycle. Both the RISC-V S-type and B-type formats have three input operands, as seen in Figure 2.2. It should be noted that when the S- or B-format operation is scheduled, the core could be transporting a result from a previous operation in the result bus, which means that the result bus cannot be used for the transportation of one of the input operands. This dictates that the core must have the minimum of four busses to be able to transfer three input operands and the result from the previous operation in one clock cycle.

RISC-V instruction set architecture specifies the size and the width of the register file. To match the programming interface of the RISC-V instruction set architecture, the underlying TTA microarchitecture must also have a register file with at least 32 entries to implement the RV32I subset or 16 for the smaller RV32E subset. The size is not however the only requirement for the register file. Many of the RISC-V instruction formats perform two register file reads while performing one write to the register file in the write back stage. To support this operation model, the register file must have two read ports and one write port.

ADF allows the user to freely modify the interconnect connectivity of the generated cores. The connectivity plays a crucial role in the compilation as it dictates which moves between sources and destinations are supported by the core. To support the load-store model, the core must have connectivity from the register file output ports to all register operand input ports. Respectively, all operation output ports must have connectivity to the register file input ports. However, it is not sufficient that the core has the required connectivity, it must also be able to schedule all the required moves in parallel. This way the transport busses are mapped to their dedicated register and immediate operands. In RISC-V's case, the core has dedicated busses for rs1, rs2, rd and immediate operands.

A minimal configuration to implement the RISC-V ISA is described in Figure 4.4. The configuration has the required two register file read ports and one write port. The interconnect has the four busses that are mapped to RISC-V operands. As seen in the figure, the order of the operand busses does not matter as long as they have the required connectivity. The rs2 operand bus is the uppermost bus and below that is the rs1 bus. The input operand busses rs1 and rs2 have only connections from the register file output to function unit input ports. Respectively, the result operand bus rd only has connections from the function unit output ports to the register file input port. The immediate bus is a special case as its operand is expressed directly in the RISC-V instruction word.

The microarchitecture has two separate function units: M that contains both the load-store and arithmetic logic unit and the C that is the control unit. The function unit M has three input ports to support the RISC-V S- and B-format, with three input operands. As seen in the figure, the immediate operand bus has connections to two input ports in the M function unit. This connectivity requirement is caused by the I-format that has shared operations with the R-format, but the rs2 operand is replaced with the immediate operand [6]. This is why the immediate operand must have the same connectivity as the rs2 operand to the function unit that implements the I-format operations. In addition to this, the S-format requires the transportation of both the rs2 and immediate operand, which is why the immediate operand must also have its own input port for the function unit that implements the S-format operations.

The control unit also has three input ports to support the three input operands of the B-format. The control unit, however, has two output ports. The reasoning behind this is that the return address is implemented as a special register in OpenASIP's TTA implementations. The operation model of function calls was not changed for this work and the return address operand was implemented as a special register port and the AUIPC result operand as its own. The ideal solution would be that the control, load-store and arithmetic logic unit were implemented as one function unit for scalar implementations to maximize the amount of hardware sharing between operations and to minimize the bypass logic. However, there is a current hardware generator limitation in OpenASIP that disables the combining of control unit with other functions units.

4.4 Hardware Generation

When generating the RISC-V microcode layer for the internal TTA microarchitecture, the hardware generator needs to have information about the internal TTA core's programming interface. OpenASIP describes the information needed to program the generated TTA processor in machine and binary encoding objects, which is sufficient to generate a RISC-V microcode layer. The machine object describes relevant information on the operations, operation latencies, function units, register files and bus configurations. The binary encoding object is relevant when generating the final program images because it has information on how the operation encodings are mapped. The microprogramming hardware can be solely generated from information acquired from the machine and binary encoding map objects.

The program flow of generating the microcode unit hardware is described in Figure 4.6. During the hardware generation, the software makes sure the internal TTA microarchitecture meets all of the design requirements for the RISC-V mode. First, the software checks that the microarchitecture supports all the required operations. If this condition is not met, the software throws an error which notifies the user that the RISC-V mode cannot be generated for the given TTA core.

ADF allows the user to freely map the operation operands to different ports in the function unit. This must be taken into account because in TTA's programming model, the moves are assigned to and from ports. This is especially important for operations whose behaviour is dependent on the operand mapping. RISC-V has four different operands: rs1, rs2, immediate and rd. Because the behaviour of the operations are fixed both in the RISC-V specification and in OpenASIP's operation models, the software can map operation ports directly to four different maps. This way it can be deducted on operation level which function unit port corresponds to which RISC-V operand.

In the next step, the register file information is analyzed. If the register file does not have enough entries to meet the RVI or RVE specification, an error is thrown. For the scheduling to work, the register file must have the minimum of two read ports and one write port. This property is also checked in this step.

Register file ports are not bound to any operation operands so they can be connected to the busses in any way possible as long as write ports are used as input and read ports as output. It is up to the hardware generation software to decide how the register file ports should be mapped to the operand busses so that the scheduling is possible. Connecting the register file ports and bussed to operands depend on each other because the register file connectivity has an effect on which busses can be used for the transportation of which operands. The finding of the operand busses is purely iterating the different combinations of the register file ports and the transport busses because all the operation operands are mapped to function unit ports because of fixed behaviour of the operations.

The algorithm that maps the operand busses is described as pseudocode in Figure 4.7. In practice, the algorithm works as a microscheduler that deducts how the available busses and register file ports should be mapped between operands so that the operands can be

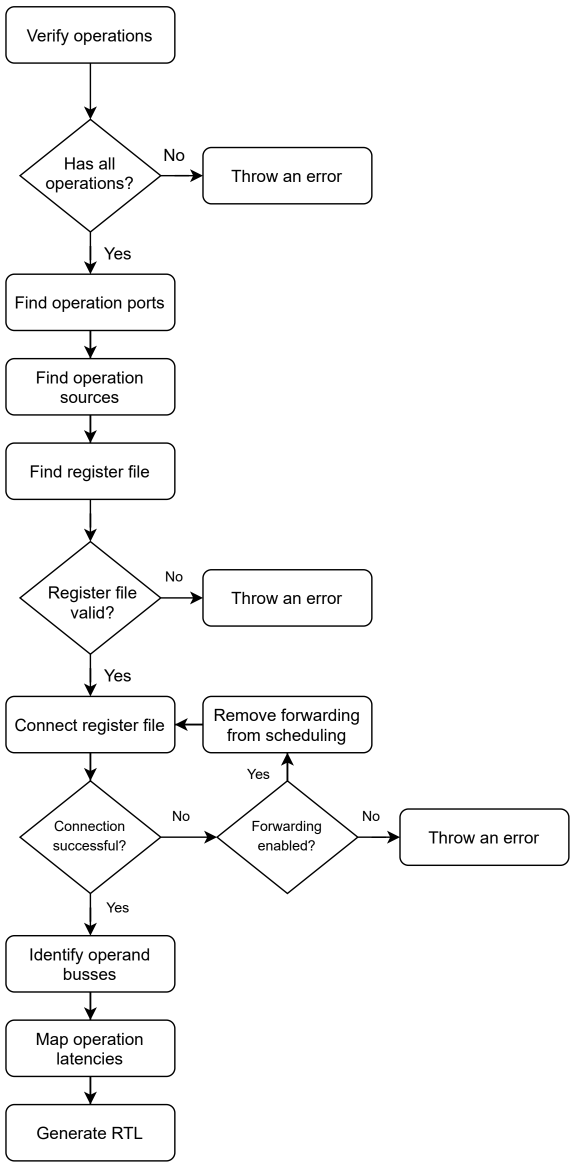

Figure 4.6. *Overview of the microcode unit generation*

transported in parallel by utilizing the available connectivity in the interconnect. When the algorithm is run for the first time, it attempts to connect the busses and register file ports with data forwarding enabled. If the algorithm is unable to find the operand busses, data forwarding is removed from the scheduling and the iterations are run again.

The algorithm is separated into two different functions. At the start of the first function, every output port is added to the list of input operand ports if data forwarding is enabled. This way the bypass connectivity can be verified by checking the connectivity between function unit input and output ports on the register operand bus. Then the register file ports are iterated in the three for loops that check every possible combination of register file ports and operands. During each iteration the second function is called, which finds out if the operand busses can be assigned to that particular register file port combination. The suitability of the busses for the tested operand is evaluated by inspecting if the interconnect is able to move data from every source port to every destination port on that bus. Additional inspection must be made for the immediate operand as its bus must be able to transport an immediate value. After every iteration is run, the algorithm has either found suitable operand busses for each RISC-V operand or concluded that suitable configuration cannot be found for the microarchitecture.

If the finding of operand busses was successful, the software can inquire all relevant information that is needed to instantiate the microcode hardware. The hardware must be aware of the exact places of the operand move slots in the instruction word as well as the places of register indexes so that they can be directly mapped between RISC-V and TTA instructions.

To support varying length operation latencies between operations, the microcode hardware must be able to identify each operation's latency. During the next step, a lookup table is generated for the supported operations.

Finally, when all the relevant information has been inquired, an instruction lookup table that maps RISC-V operation code and function fields to TTA instructions can be generated by first generating instruction bits for the underlying TTA microarchitecture and then storing them in a hardware lookup table.

4.5 Customization Points

The key benefit of generating the hardware descriptions in software from an architectural level description is that it allows many and complex customization points without using complex if-generate and generic structures in the hardware description. The customization points implemented in this work are presented in Table 4.1.

The current implementation supports three and four pipeline stages. To support the configuration of three pipeline stages, the instruction register in the instruction fetch unit must be disabled. In this configuration, the fetching, translation and decoding of the instruction are done in the same combinatorial path. As described in Figure 4.1 the execute and register file read are separated from the writeback into their own stage.

The extensions are automatically configured from the definition of the architecture. If the architecture has at least 32 register file indexes, the base integer subset is automatically picked. The smaller RV32E subset is chosen if the architecture has only 16 register file indexes. The optional M extension is generated if the required operations are defined in the ADF.

Operation latencies can be fully customized. From the front end point of view, only the scheduling of the result move must be reconfigured based on the defined operation latencies, which is implemented by storing the operation latency as an entry into a lookup table.

```
function ConnectRF
    if dataForwarding then
        rs1Ports.add(rdPorts)
        rs2Ports.add(rdPorts)
    ports.rs1 ← rs1Ports
    ports.rs2 ← rs2Ports
    ports.rd ← rdPorts
    ports.imm ← immPorts
    for all w1 ∈ rfWritePorts do
        ports.rd ← rd.Ports
        ports.rd.add(w1)
        for all r1 ∈ rfReadPorts do
            ports.rs1 ← rs1Ports
            ports.rs1.add(r1)
            for all r2 ∈ rfReadPorts do
                ports.rs2 ← rs2Ports
                if ports.rs1.hasElement(r2) then
                    continue
                ports.rs2.add(r2)
                if connectBusses(ports, operandBusses) then
                    return true
    return false

function connectBusses(p b)
    for all bus ∈ busses do
        if not isConnected(p.simm, bus) || not bus.hasSimm then
            continue
        b.simm ← bus
        for all bus ∈ busses do
            if not isConnected(p.rs1, bus) || b.hasElement(bus) then
                continue
            b.rs1 ← bus
            for all bus ∈ busses do
                if not isConnected(p.rs2, bus) || b.hasElement(bus) then
                    continue
                b.rs2 ← bus
                for all bus ∈ busses do
                    if not isConnected(p.rd, bus) || b.hasElement(bus) then
                        continue
                    b.rd ← bus
                    return true
    return false
```

Figure 4.7. *Algorithm for finding operand busses*

The control logic stalls the pipeline until the multi-cycle operation has been executed. The implementation could be improved by allowing the pipelining of operations, but that would require more complex data hazard control as the hardware would have to keep track of the operations in the operation pipeline to forward the correct result operand.

Bypass connection customization is partly supported. The data forwarding can either be entirely disabled or enabled. During the generation of the microprogramming hardware, the software checks whether the interconnect has connectivity for all the bypass moves. If the core supports bypass moves to all register operand input ports, the bypass logic is generated to the microcode hardware. Customization of the bypass network was not found in other RISC-V generators. In this work, it is easy to modify the bypass network due to the use of exposed datapath processor as the internal microarchitecture.

Pipeline stages	3-4
Extensions	RV32E/RV32I(M)
Operation latencies	Fully customizable
Bypass connections	Partly customizable

Table 4.1. *Customization points of the generated RISC-V cores*

5. VERIFICATION AND EVALUATION

This chapter explores the ways the generated RISC-V cores were evaluated and verified. First, the implementation details of the generated cores are compared against two open source RISC-V implementations. Then the post-synthesis area and maximum clock frequency are evaluated and, after that, the performance figures are presented. Then the verification methods that were used to verify the hardware are discussed.

5.1 Reference Implementations

The generated RISC-V cores were evaluated against two RISC-V implementations: zero-riscy [34] and RI5CY [35]. However, there are differences in the cores' implementation, which are explained in this section.

The operation latencies are described in Table 5.1. The values in the table are best-case latencies, as data hazards and dynamic latencies can cause additional stalls when the operation is executed. One of the major differences between the implementations is in how the memory operations are handled: zero-riscy requires a minimum of two cycles to finish both load and store operations whereas the generated and RI5CY cores can execute load and store operations without any additional stalls. Zero-riscy does not bypass the load data and instead always loads it into the register file before resuming execution, even without a data hazard. The stalls during store operations are caused by waiting for a response from the memory before resuming execution [36][37]. Zero-riscy's handling of memory operations causes additional overhead, especially in data-oriented code.

Pipelining has an effect on the implementation of control flow operations. Due to its shorter pipeline, zero-riscy has a smaller branch penalty compared to the generated and RI5CY cores, which can be seen as a reduction of one clock cycle in the latency of taken branch operations [36][37]. The generated cores do not implement flushing for branch operations, which means that the branch is never predicted taken and instead all branches are executed the same way, independent of whether they are taken or not.

Another difference is in the pipelined mulh, division and remainder operations. Remainder and division are complex operations that in zero-riscy's and RI5CY's case are implemented with dynamic latencies where the latency of the operation is dependent on its input values [36][37]. Dynamic latencies were not implemented for the generated cores and instead the latencies for such operations were fixed to static latency. The support for dynamic latencies could be added by assigning the operation latency to the best-case value and locking the core pipeline from inside the function unit until the operation has been executed if the latency exceeds best-case value.

The RI5CY core differs from the two cores significantly because it has additional custom features that are not implemented for zero-riscy or the generated cores. These features include support for hardware loops, optional FPU support and custom instruction-extensions, such as dot product operations. RI5CY has also a more complex register file compared to the other implementations. RI5CY has a register file with two write and three read ports instead of one write and two read ports like in zero-riscy and the generated cores. The more complex register file is due to the load-store unit having its own write port and the extra read port that was added for custom operations, such as multiply and accumulate. [34][35][36][37]

Reference implementations also handle data hazards in a different way. RI5CY core experiences a penalty of one clock cycle for data hazards originating from the load-store unit output due to the missing bypass connection from the load-store unit read data [37]. Zero-riscy's data forwarding was not documented, but data hazard related stalls were not witnessed in the execution traces.

Three configurations were generated to evaluate properties of the generated cores: three-stage versions with and without bypass support as well as four-stage configuration with bypass support. The architecture definition of Figure 4.3 was used to generate the cores, but the bypass connections were removed for the core that has no bypass connectivity. In the implementation, the LSU and ALU were implemented as one function unit, while the control unit that contains the control flow operations was implemented as a separate function unit.

A system on a chip implementation, Pulpino [38], was used to integrate and benchmark the zero-riscy and RI5CY cores. Pulpino separates the instruction and data memories and connects them to an AXI4 interface, which allows the cores to access both memories either locally or via the AXI4 interconnect [39]. This allows the cores to run self modifying code. The generated cores also follow the Harvard architecture where the instruction and data memories are separated, but the current test bench does not allow the LSU interface to access the instruction memory, which disables the use of self modifying code. This limitation is purely due to the integration of the memories in the test bench and not a feature of the generated cores. The cores were not, however, tested with self modifying code.

Operation Type	zero-riscy	RI5CY	3 stages	4 stages
Integer Arithmetics	1	1	1	1
Load/Store	2	1	1	1
MUL	1	1	1	1
MULH	2	4	4	4
Division/Remainder	1 or 37	2-32	35	35
Branch (Not-Taken)	1	1	3	4
Branch (Taken)	2	3	3	4
Jump	2	2	2	3

Table 5.1. *Operation latencies of the cores*

5.2 Synthesis Results

Post-synthesis properties of the cores were evaluated by synthesizing the designs with Synopsys Design Compiler [40] and a 28nm technology without memories. Zero-riscy and the generated cores were configured without the M extension to highlight the effect of the microcode hardware and to remove the effect of the different implementation of the area consuming M extension. Also, the debugger, control and status registers, prefetch buffer and the compressed decoder were removed from zero-riscy's RTL. Due to the more complex structure of the RI5CY and the non-configurable M extension, the RI5CY core was synthesized without making any modifications to the RTL, which makes RI5CY's synthesis results not directly comparable with the other designs.

5.2.1 Comparison Against Reference Implementations

All of the designs were synthesized with their maximum clock frequencies as their timing target. The design areas are described in Figure 5.1. Zero-riscy achieves the smallest area compared to other implementations. The four-stage configuration utilizes approximately 13% more area than zero-riscy and is the biggest in terms of area of the generated cores which is expected due to the extra pipeline registers. Comparison of the three-stage configurations gives an idea of the area overhead of the bypass connectivity. The configuration with no bypass connectivity achieves the lowest area of the three generated cores even though it utilizes 7% more area than zero-riscy. The version with three pipeline stages and full bypass connectivity utilizes 11% more area than zero-riscy. Of the customization points used in the synthesis, the bypass connectivity has the biggest effect on the design area, causing approximately 4% addition to the area utilization. RI5CY has 290% bigger area than zero-riscy due to its extra features that are not implemented for the other designs.

Some of the area overhead compared to zero-riscy is caused by the extra pipeline stage that adds registers to the execute stage. Additional overhead is caused by OpenASIP's current hardware generator limitation that does not allow to combine the control unit with other function units as seen in Figure 4.3. This makes indicating resource sharing between operations difficult in the hardware description. Additionally, it adds extra registers to the design as each function unit output port has its own registers. Additional complexity is also added to the forwarding logic by the separate function units because of the increased amount of bypass path combinations.

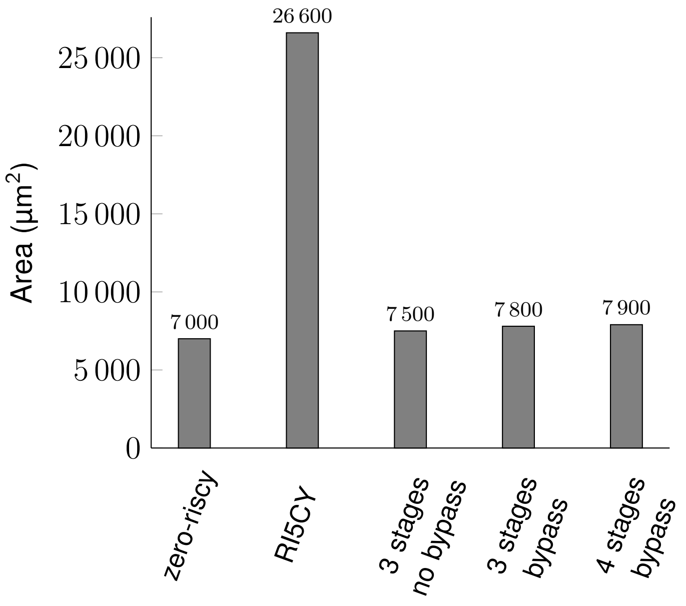

Figure 5.1. *Area utilization of the synthesized cores*

The maximum clock frequencies of the designs are described in Figure 5.2. The generated designs achieve the highest maximum clock frequency of the cores. The four and three-stage versions achieve the same clock frequency. In all the generated designs, the critical path started from the decode output registers and went through the register file read, interconnect and ending in the program counter register in the instruction fetch unit. In the four stage version, the instruction fetch was separated into its own stage. The synthesis tool was unable to retime the registers and therefore both the critical path and timing were the same between the pipeline configurations. However, the extra pipeline register in the instruction fetch could prove useful when memories are connected. The bypass connectivity had only a small impact on the maximum clock frequency, making the version without bypass connectivity 2% faster than other generated designs. Even the versions with bypass support achieve significantly higher clock frequencies than the two reference designs, beating zero-riscy by 20% and RI5CY by 63%.

The generated cores have one or two more pipeline stages than zero-riscy which explains the higher clock frequency. The RI5CY core is slow compared to the other designs because of its more complex operation set, which causes a long combinatorial path through the execute stage. It should be noted that the synthesis was run with the core as the top level design, which excludes memories. If the memories were connected, the design could have a different maximum clock frequency and critical path. This would most heavily impact the LSU, as it has a long combinatorial path before the actual memory access.

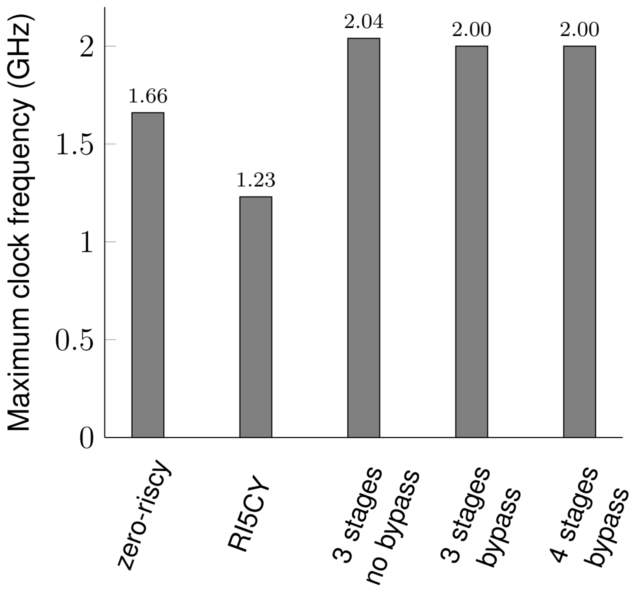

Figure 5.2. *Maximum clock frequencies of the synthesized cores*

5.2.2 Overhead Evaluation

The three-stage version with bypass support was also evaluated against a similar TTA core without the RISC-V front end and the required changes to the instruction fetch unit. TTAs use shadow registers in function unit ports to store data between clock cycles. However, in the RISC-V mode these shadow registers can be removed because all input values are transported during the same clock cycle. The shadow registers were also removed from the TTA core to help pinpoint the overhead of the microcode hardware.

The static timing analysis revealed that both the core with and without RISC-V front end achieve the same clock frequency of 2.0 GHz, which is expected as the microcode hardware is not on the critical path of the design. The TTA design achieved almost identical area to the RISC-V implementation with these settings. However, the instruction fetch units are not identical between the implementations. The RISC-V implementation routes some control flow operations directly to the instruction fetch unit and the TTA has a wider instruction word, which affects the area results. With flattening disabled, the TTA core utilized 2.9% less area than the RISC-V core, which indicates that the flattening helps in reducing of the microcode area.

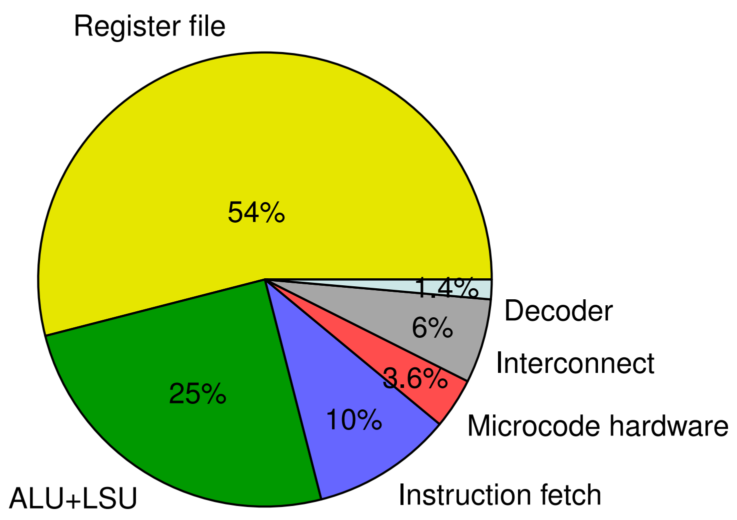

Figure 5.3. *Break down of the area utilization of a generated core with 3 stages and full bypass connectivity*

The breakdown of the area utilization between different components is presented in Figure 5.3. The microcode hardware takes approximately 3.6% of the design area, while the register file combined with the function units and instruction fetch unit take 89% of the total area. The lookup tables that are used for the bypass and register moves only consisted 1% of the design area. It should be noted that to evaluate the area of different components, flattening had to be disabled, which has an impact on the results as the synthesis tool cannot optimize the design through hierarchies.

In combination, the decoder and microcode hardware utilized 5% of the design area, which is close to the relative decoder utilization of zero-riscy which was 4.5%. More accurate results of the area utilization of the control and decode logic could be acquired if the microprogramming hardware was implemented inside the decoder component instead of as a separate component. This, however, makes hardware generation more complex and should not yield better results because when the design is flattened, the synthesis tool is able to optimize the design through hierarchies.

5.3 Performance

All the RISC-V cores were benchmarked with a benchmark suite CHStone [41] that does not have benchmarks with floating point computation. The source code was compiled with RISC-V GNU Compiler Toolchain [42] version 11.1.0 that was configured to the RV32IM subset of the RISC-V ISA. JPEG as well as the double precision benchmarks were left out because they were too large for Pulpino's 32kB memories.

The cycle count comparisons are presented in Figure 5.4. The generated cores with bypass support achieve significantly better cycle counts than the two reference cores.

The three-stage configuration has approximately 28% and 16% lower cycle counts than zero-riscy and RI5CY on average. The configuration with four pipeline stages suffers an overhead from the higher control flow latencies and achieves only 24% and 12% lower clock cycles compared to zero-riscy and RI5CY. The effect of missing bypasses can be seen in the cycle count results of the configuration with no bypass support that has on average 11% higher cycle counts than zero-riscy and 56% higher than the three-stage version with bypass support. The bypass connectivity is a very useful feature as it has only a minimal effect on the synthesis results while lowering the cycle counts significantly.

However, the RI5CY core suffers a significant overhead from the missing forwarding support from the load-store unit, causing a 22% overhead on average. Zero-riscy suffered even larger overhead due to the multi-cycle memory operations where both load and stores take the minimum of two cycles to complete. The overhead of the stalls caused by memory operations in zero-riscy's case was 47% in average.

Additional distortion in the results is caused by the differences in the division and remainder operations that are implemented with dynamic latencies in the PULP-based reference cores. However, these are relatively rare operations in the used benchmarks and only used in the aes benchmark where they make approximately 0.2% of all executed instructions. The mulh operations was not used in the benchmarks.

If the stalls caused by the load-store unit are discarded, the configuration with three pipeline stages and bypass support has 5% and 2% higher cycle counts compared to zero-riscy and RI5CY. Zero-riscy can handle control-oriented code with less penalty due to its less aggressive pipelining. The overhead caused by the missing flush support can be seen in the control heavy mips benchmark where not taken branches make 3.5% of all instructions. In this scenario, the configuration with three pipeline stages and bypass support has 11% and 6% higher cycle counts than zero-riscy and RI5CY if the load-store unit caused stalls are removed. Even though the two generated cores are able to achieve low cycle counts due to their memory interface and bypass connectivity, the evaluated penalty of control flow operations adds motivation for future flush support.

Run time of the benchmarks described in Figure 5.5 can be extracted by combining the clock cycle counts presented in Figure 5.4 with the maximum clock frequencies presented in Figure 5.2. The configuration with three pipeline stages and bypass support achieves the lowest run time of the designs because of its higher clock frequency, offering in average 41% and 48% lower run time than zero-riscy and RI5CY. Even when the load-store unit caused stalls are removed from the run time, the cores's run time is 13% and 37% lower than zero-riscy's and RI5CY's. The configuration with four pipeline stages achieved 8% and 34% lower run time when the load-store unit induced stalls were discarded.

Overall, the reference cores and the configuration with three pipeline stages and bypass support achieved similar clock cycle counts when the LSU induced stalls were discarded

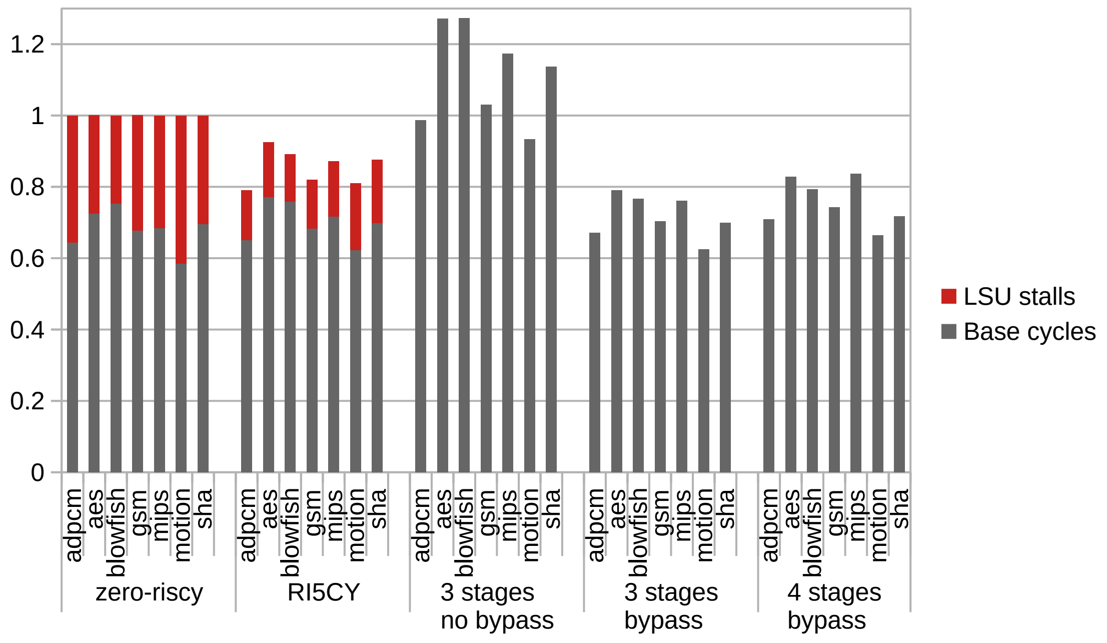

Figure 5.4. *Cycle counts compared to zero-riscy baseline*

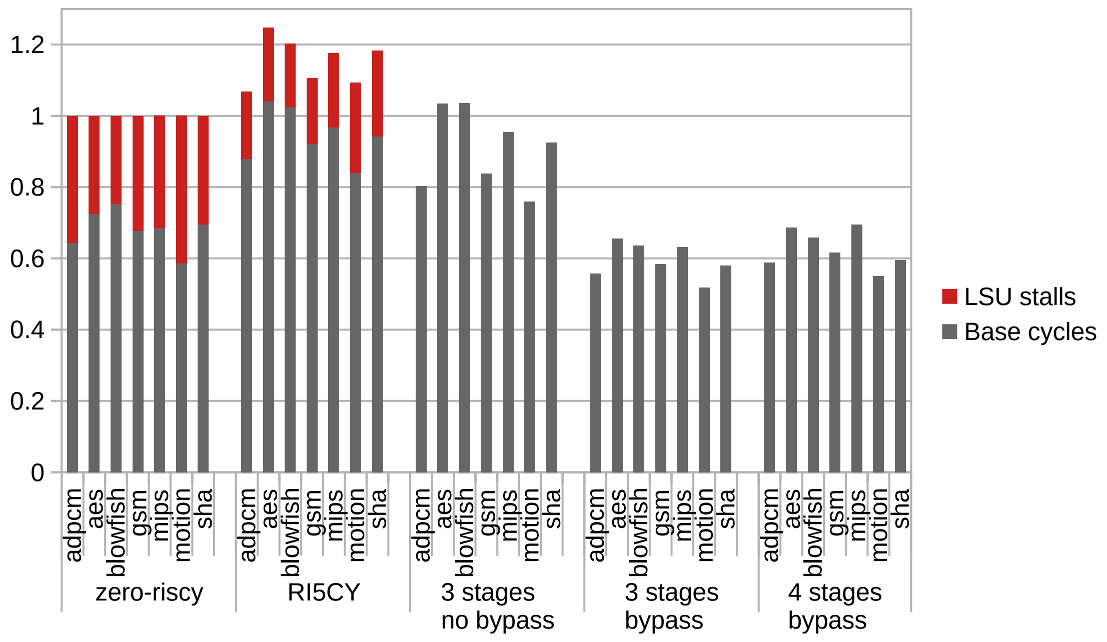

Figure 5.5. *Run time compared to zero-riscy baseline*

from the results of the reference cores. This is due to the similar operation latencies between the cores, which proves that using a TTA core as the internal microarchitecture to implement the RISC-V ISA can provide competitive results, even providing significantly faster run times than the two reference implementations. The configuration without bypass connectivity experiences significant overhead due to the missing operand forwarding support, which emphasises the importance of bypass connectivity. The configuration with an extra pipeline stage could not achieve better performance as it had higher control flow latencies and same clock frequency as the three pipeline stage version. Better clock frequencies could be achieved if the register file read was made synchronous or registers were added to the function unit input ports. If the register read were synchronous, the microcode would need a more complex sequencer as the micro-operations would need to be sequenced in three cycles instead of two as in the current implementation. Registers in the function unit input ports would effectively raise the operation latency to two clock cycles. It could, however, be trivially pipelined, but that would cause problems with data forwarding as it has a similar effect to separating the memory access to its own stage where even a forwarded operand would cause a stall during a data hazard.

5.4 Verification

Verification is an important step of making sure that the hardware design meets the requirements set in the specification. When designing processor cores, the specification is the instruction set architecture. A common way to verify that the design works correctly is feeding the design stimuli in a hardware test bench and verifying that the design emitted the correct output. However, just verifying the output was correct does not give a strong coverage on the operation of the internals of the hardware. A more sophisticated way to verify hardware is to emit traces on the execution and compare them against *golden traces*, which also helps debugging the hardware in case the design does not work as expected by the specification.

The verification structure used to verify the hardware in this thesis work is described in Figure 5.6. The test programs shown on the left of the figure consist of the CH-Stone benchmark suite that includes 12 programs written in the C programming language. RISC-V GNU Toolchain was used to compile and generate the program images from the source code. The program images were read in the hardware test bench and uploaded into the instruction and data memories of the core. The design source files were compiled and simulated with a register transfer level simulator Modelsim [43]. To generate golden traces, a RISC-V instruction set simulator riscvOVPsim [44] was used. In the final step, the golden traces output by riscvOVPsim were compared against the ones generated during RTL simulation. The generated verification traces were divided into two different types: instruction traces and datapath traces.

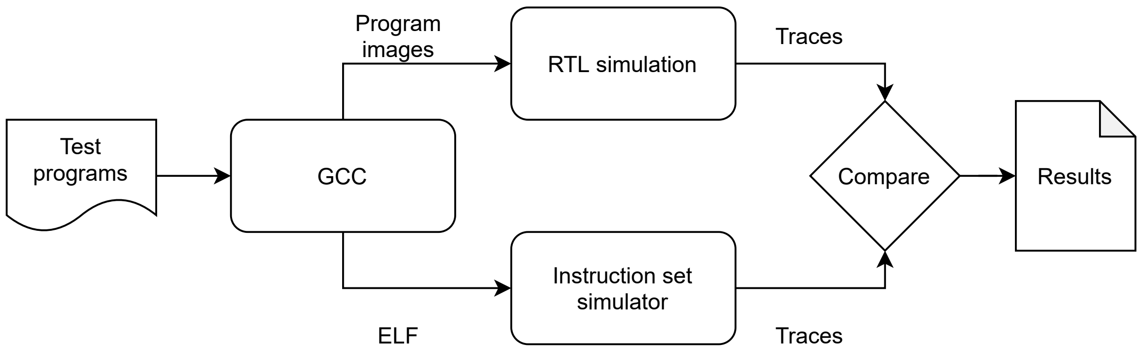

Figure 5.6. *Overview of the hardware verification structure*

Comparing the log of executed instructions against a golden trace gives a good view of the control flow of the hardware. This way, the test environment can verify that the core executed the correct instructions during program runs. However, it by itself does not verify that the core created the correct output and executed the operations correctly when they do not affect the control flow directly.

Generating instruction traces during program runs is simple. In this work, the instruction tracer was placed in the microcode hardware. The tracer works by writing a new entry into a text file every time there was a new incoming RISC-V instruction that was not bubbled. Cores with flushing support would need additional logic to the trace generation because in their case, not all decoded instructions are executed.

Datapath verification gives a strong coverage of the internal workings of a processor core. In transport triggered architectures, this must be done by emitting the data that is being transported in the interconnect busses because not all data is routed through the register file. In load-store architectures, it is sufficient to emit the data that is written into the register file. When both the instruction trace and the datapath trace are compared against the equivalent golden traces, it can be deducted whether the design executed the program correctly.

During verification the core received 86% total line coverage while the microcode hardware received 66% line coverage. The tested benchmarks did not utilize all the operations and the bypass paths, which affected the acquired coverage as many of the entries in the instruction and bypass lookup tables were not tested. When the lookup tables were excluded, the microcode hardware received 96% line coverage. More extensive verification could be added by running unit tests on the core, which would stress all the operations and bypass combinations.

6. FUTURE WORK

This chapter discusses the additional features and improvement points that could be implemented to extend the work initiated in this thesis. RISC-V is an extendable and customizable ISA, which enables a lot of extra features that were not implemented in this work but would allow more customization points to the RISC-V generator presented in the thesis.

6.1 Pipeline Flush Support

Flushing of instructions from the processor pipeline after a taken branch is a common optimization in processor implementations. In practice, the core pipeline would need some sort of guard that would disable writes to the register file and execution of store operations if the instructions in the pipeline are invalid. This way, no modifications to the program state would not be made by invalid instructions.

Currently, the RISC-V cores generated in this work do not have support for flushing of instructions which caused a small overhead, especially in control oriented code MIPS benchmark. Additional changes in the hardware generation would need to be made to implement flushing. Most importantly, it would have to be inspected if the conditional branches are taken and then disable writes to the register file and memory until the invalid instructions have been flushed from the pipeline. Another way to implement flushing is to use the trigger guards that disable the execution of an operation in transport triggered architectures based on a boolean register value. This way, the boolean register could be set to disable operation triggers while there are invalid instructions in the pipeline and therefore disable any unwanted changes to the program state.

6.2 64-bit Instruction Set and Additional Extensions

The current implementation allows to generate the RV32E/32I(M) configurations. The RISC-V ISA includes lots of different standard extensions such as atomic, floating, vector as well as control and status operations. They could be included the same way to the microcode as was done for the M extension in this work, where a map of the instruction bits and the operations of the extension are added to the hardware generation to allow the generation of the microcode. Then, if the defined architecture has all the required operations, the extension would be automatically generated. Especially interesting extension is the vector extension that can be added to exploit data-level parallelism.

The 64-bit subset of the RISC-V ISA could be easily added because it only requires the extension of the core's datapath to 64 bits and a few additional operations. OpenASIP has partial support for 64 bit designs that could be reused for the 64-bit subset of the RISC-V ISA.

6.3 Custom Operations

Support for custom operations was a popular as well as important customization point in the commercial RISC-V tools. In the scope of this thesis work, the support for hardware generation of custom operations could be added effortlessly. The free opcodes in the RISC-V ISA could be used to declare encodings for the custom instructions and then add them to the microcode of the RISC-V front end. However, having only the hardware with the custom operations is not by itself very useful unless the programmer wants to only directly invoke them with machine code.

The most major issue with custom operations is that they would require compiler support. Some implementations fork their own version of the RISC-V GCC Toolchain and add the support for their own custom extensions to the compiler backend as was done for the RI5CY [35]. However, this is not a very sophisticated solution for design exploration as the user would have to manually extend and recompile the compiler every time a new custom operation is added. A solution for this is to use a retargetable compiler that would allow the addition of operations into the compiler without the need to recompile the compiler.

OpenASIP's retargetable LLVM-based compiler tcecc could be extended to allow compilation for different subsets of the RISC-V ISA by putting restrictions to the architecture description, which would effectively produce RISC-V code. A key feature of this is to restrict the architecture to operation triggered mode where all input operands are transported during the same cycle and the result operand is always written to the register file, which would disable the use of software programmable bypasses. Then the compiler generated output could be transformed to RISC-V binaries by mapping them during program image generation.

7. CONCLUSIONS

This master thesis work implemented a RISC-V generator by extending the OpenASIP toolset. The generator works by utilizing a TTA core as the internal microarchitecture together with a generated microcode unit that implements the operation triggered features as well as part of the control logic. Microprogramming has been extensively used to implement control units for CISC processors but not for RISC implementations, which adds novelty to this work and enables the reuse of features from the OpenASIP tool flow for RISC-V customization.

The current implementation allows the customization of the amount of pipeline stages, operation latencies, addition of the M extension and partial customization of the bypass network. Support for custom operations was a key feature in commercial RISC-V generators. However, it was not added in this work as it needs compiler support, which is out of scope for the topic of the thesis. Additional interesting future work is the addition of extra standard extensions of the RISC-V ISA and the extending of the subset to 64 bits.

In the evaluation section, three generated RISC-V designs with different customization options were evaluated against two open source implementations: zero-riscy and RI5CY. Configuration with 3 pipeline stages and bypass support achieved the best performance of the generated cores and reached 13% and 37% lower run time compared to zero-riscy and RI5CY when the cycles caused by differences in the implementation of load-store units were discarded. The generated cores achieved higher clock frequencies than the reference implementations but consumed more area than zero-riscy. Some area overhead is caused by OpenASIP's current hardware generation limitation that prevents the fusing of the control unit with other function units. The microcode hardware itself utilized only 3.6% of the design area.

The evaluated penalty in control-oriented code caused by the missing pipeline flush support adds motivation for future work on optimizing the pipeline to allow pipeline flushes. The more deeply pipelined configurations could be optimized more by moving the pipeline stage registers from the instruction fetch unit to either the register file output or to function unit input ports, as those are in the critical path of the design. Additional critical path analysis should be performed with integration to memories or caches so that the critical paths are analyzed in a more realistic setup.

REFERENCES

[1] Dandamudi, S. P. *Guide to RISC Processors for Programmers and Engineers*. 1st ed. 2005. New York, NY: Springer New York, 2005.

[2] Patterson David A. Hennessy, J. L. *Computer organization and design: the hardware/software interface*. Morgan Kaufmann, 2013.

[3] Patterson, D. A. and Ditzel, D. R. The case for the reduced instruction set computer. *Computer architecture news* (1980).

[4] Wilkes, M. V. The best way to design an automatic calculating machine. *Proc. Manchester Univ. Computer Inaugural Conf.* 1951.

[5] Hennessy, J. L. and Patterson, D. A. *Computer Architecture: A Quantitative Approach*. Elsevier Science & Technology, 2011.

[6] Waterman, A. and Asanovíc, K. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2*. Tech. rep. 2017. URL: `https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf`.

[7] Waterman, A. S. *Design of the RISC-V Instruction Set Architecture*. eScholarship, University of California, 2016.

[8] Lee, D., Kong, S., Hill, M., Taylor, G., Hodges, D., Katz, R. and Patterson, D. A VLSI chip set for a multiprocessor workstation. I. An RISC microprocessor with coprocessor interface and support for symbolic processing. *IEEE Journal of Solid-State Circuits* (1989).

[9] *Processor Design System-On-Chip Computing for ASICs and FPGAs*. 1st ed. 2007. Springer Netherlands, 2007.

[10] Hoogerbrugge, J. and Corporaal, H. Transport-triggering vs. operation-triggering. *Compiler Construction*. Ed. by P. A. Fritzson. Springer Berlin Heidelberg, 1994.

[11] Corporaal, H. *Microprocessor Architectures: From VLIW to TTA*. John Wiley & Sons, Inc, 1997.

[12] Leupers, R., Deprettere, E. F., Bhattacharyya, S. S. and Takala, J. *Handbook of Signal Processing Systems*. Springer, 2018.

[13] *Designing Embedded Processors A Low Power Perspective*. 1st ed. 2007. Springer Netherlands, 2007.

[14] Nohl, A., Schirrmeister, F. and Taussig, D. Application specific processor design: Architectures, design methods and tools. *2010 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2010.

[15] Keutzer, K., Malik, S. and Newton, A. From ASIC to ASIP: the next design discontinuity. *Proceedings. IEEE International Conference on Computer Design: VLSI in Computers and Processors*. 2002.

[16] Mishra, P. *Processor description languages applications and methodologies*. 1st edition. Morgan Kaufmann Publishers/Elsevier, 2008.

[17] Hoffmann, A., Meyr, H. and Leupers, R. *Architecture exploration for embedded processors with LISA*. Vol. 3. Springer, 2002.

[18] Jääskeläinen, P., Viitanen, T., Takala, J. and Berg, H. HW/SW Co-design Toolset for Customization of Exposed Datapath Processors. *Computing Platforms for Software-Defined Radio*. Ed. by W. Hussain, J. Nurmi, J. Isoaho and F. Garzia. Springer International Publishing.

[19] *TTA-based Co-design Environment User Manual v1.22*. Tampere University, 2020. URL: http://openasip.org/user_manual/TCE.pdf.

[20] *Codasip Studio*. Codasip. URL: https://codasip.com/.

[21] *Codasip Studio brochure*. Codasip. URL: https://news.codasip.com/brochure/.

[22] Wargéus, P. and Forsberg, L. *Customized Processor Design for 5G Data Link Layer Processing*. eng. Student Paper. 2020.

[23] Eissa, A. S., Elmohr, M. A., Saleh, M. A., Ahmed, K. E. and Farag, M. M. SHA-3 Instruction Set Extension for A 32-bit RISC processor architecture. *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 2016.

[24] *SiFive Core Designer*. SiFive. URL: https://www.sifive.com/core-designer/.

[25] *Andes*. Andes Technology. URL: https://www.andestech.com/.

[26] *ASIP Designer*. Synopsys. URL: https://www.synopsys.com/dw/ipdir.php?ds=asip-designer.

[27] *ASIP Designer Datasheet*. Synopsys. URL: https://www.synopsys.com/dw/doc.php/ds/cc/asip-designer-ds.pdf.

[28] *MP Designer*. Synopsys. URL: https://www.synopsys.com/dw/ipdir.php?ds=mp-designer.

[29] *WARP-V*. Redwood EDA. URL: https://github.com/stevehoover/warp-v.

[30] Asanović, K., Avizienis, R., Bachrach, J., Beamer, S., Biancolin, D., Celio, C., Cook, H., Dabbelt, D., Hauser, J., Izraelevitz, A., Karandikar, S., Keller, B., Kim, D., Koenig, J., Lee, Y., Love, E., Maas, M., Magyar, A., Mao, H., Moreto, M., Ou, A., Patterson, D. A., Richards, B., Schmidt, C., Twigg, S., Vo, H. and Waterman, A. *The Rocket Chip Generator*. Tech. rep. EECS Department, University of California, Berkeley, 2016. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html.

[31] *VexRiscv*. URL: https://github.com/SpinalHDL/VexRiscv.

[32] *SpinalHDL*. URL: https://github.com/SpinalHDL/SpinalHDL.

[33] Karaki, H., Akkary, H. and Shahidzadeh, S. X86-ARM binary hardware interpreter. *2011 18th IEEE International Conference on Electronics, Circuits, and Systems*. 2011.

[34] Schiavone, P. D., Conti, F., Rossi, D., Gautschi, M., Pullini, A., Flamand, E. and Benini, L. Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for Internet-of-Things applications. *Proceedings of International Symposium on Power and Timing Modeling, Optimization and Simulation*. 2017.

[35] Gautschi, M., Schiavone, P. D., Traber, A., Loi, I., Pullini, A., Rossi, D., Flamand, E., Gürkaynak, F. K. and Benini, L. Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2017).

[36] *Zero-riscy Documentation*. lowRISC. URL: https://ibex-core.readthedocs.io.

[37] *RI5CY User Manual*. PULP Platform. URL: https://github.com/openhwgroup/cv32e40p/blob/18c10e3659fd8b8e327843b3f4cc2ed836c1d547/doc/user_manual.doc.

[38] *Pulpino*. PULP Platform. URL: https://github.com/pulp-platform/pulpino/.

[39] *Pulpino Documentation*. PULP Platform. URL: https://github.com/pulp-platform/pulpino/blob/master/doc/datasheet/datasheet.pdf.

[40] *Design Compiler Graphical*. Synopsys. URL: https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/design-compiler-graphical.html.

[41] Hara, Y., Tomiyama, H., Honda, S. and Takada, H. Proposal and Quantitative Analysis of the CHStone Benchmark Program Suite for Practical C-based High-level Synthesis. *Journal of Information Processing* (2009).

[42] *RISC-V GNU Compiler Toolchain*. URL: https://github.com/riscv-collab/riscv-gnu-toolchain.

[43] *ModelSim*. Siemens. URL: https://eda.sw.siemens.com/en-US/ic/modelsim/.

[44] *riscvOVPsim*. Open Virtual Platforms. URL: https://www.ovpworld.org/riscvOVPsimPlus/.