An Optimized Instruction Set Architecture for AMIDAR Processors

Zur Erlangung des akademischen Grades Doktor-Ingenieur (Dr.-Ing.)
Genehmigte Dissertation von Alexander Schwarz aus Fulda
Tag der Einreichung: 29. September 2021, Tag der Prüfung: 20. Juni 2022

1. Gutachten: Prof. Dr.-Ing. Christian Hochberger
2. Gutachten: Prof. Dr.-Ing. Thilo Pionteck
Darmstadt – D 17

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Elektrotechnik und Informationstechnik
Fachgebiet Rechnersysteme

An Optimized Instruction Set Architecture for AMIDAR Processors

Genehmigte Dissertation von Alexander Schwarz

1. Gutachten: Prof. Dr.-Ing. Christian Hochberger
2. Gutachten: Prof. Dr.-Ing. Thilo Pionteck

Tag der Einreichung: 29. September 2021
Tag der Prüfung: 20. Juni 2022

Darmstadt – D 17

Bitte zitieren Sie dieses Dokument als:
URN: urn:nbn:de:tuda-tuprints-215701
URL: http://tuprints.ulb.tu-darmstadt.de/21570

Dieses Dokument wird bereitgestellt von tuprints,
E-Publishing-Service der TU Darmstadt
http://tuprints.ulb.tu-darmstadt.de
tuprints@ulb.tu-darmstadt.de

Für meinen Bruder, meine Eltern und Großeltern, die mich begleiten, seit ich denken kann.

Erklärungen laut Promotionsordnung

§8 Abs. 1 lit. c PromO

Ich versichere hiermit, dass die elektronische Version meiner Dissertation mit der schriftlichen Version übereinstimmt.

§8 Abs. 1 lit. d PromO

Ich versichere hiermit, dass zu einem vorherigen Zeitpunkt noch keine Promotion versucht wurde. In diesem Fall sind nähere Angaben über Zeitpunkt, Hochschule, Dissertationsthema und Ergebnis dieses Versuchs mitzuteilen.

§9 Abs. 1 PromO

Ich versichere hiermit, dass die vorliegende Dissertation selbstständig und nur unter Verwendung der angegebenen Quellen verfasst wurde.

§9 Abs. 2 PromO

Die Arbeit hat bisher noch nicht zu Prüfungszwecken gedient.

Darmstadt, 29. September 2021

A. Schwarz

Zusammenfassung

Java Bytecode wird häufig als binäre Repräsentation eines Programms auf vielen verschiedenen Hardwareplattformen eingesetzt. Inhärente Sicherheitseigenschaften wie das strikte Speichermodell und Codeprüfung zur Laufzeit machen ihn auch für eingebettete Systeme interessant. Java Prozessoren, die Java Bytecode direkt in Hardware ohne virtuelle Maschine und Betriebssystemschicht ausführen, versprechen in diesem Anwendungsbereich geringeren Ressourcenbedarf und schnellere Reaktion auf externe Ereignisse. Der auf Bytecode basierende AMIDAR Prozessor ist ein Beispiel für solch einen Java Prozessor. Adaptive Microinstruction Driven Architecture (AMIDAR) Prozessoren sind auf Anpassungsfähigkeit zur Laufzeit ausgerichtet. Sie bilden ohne Eingriff des Benutzers rechenintensive Codeabschnitte auf rekonfigurierbare Hardwarebeschleuniger ab. Die hohe Abstraktionsebene von Java Bytecode vereinfacht diese Abbildung, weil die ursprüngliche Absicht des Programmierers einfacher abgeleitet werden kann.

Java Bytecode ist eine Architektur, die auf einem Operanden-Stack basiert. Daten werden zwischen Instruktionen mit Hilfe eines Stacks übergeben, was viele unnötige Datenübertragungen verursacht. Der Operanden-Stack begrenzt die Ausführungsgeschwindigkeit in vielen Java Prozessoren. Diese Arbeit löst das Problem durch Entwicklung einer neuen Instruction Set Architecture (ISA) als Ersatz für Java Bytecode auf AMIDAR Prozessoren. Diese ISA eliminiert den Operanden-Stack, während sie die hohe Abstraktionsebene beibehält. Ihre Gestaltung orientiert sich an Datenflussarchitekturen. Die Instruktion, welche das Ergebnis einer Operation erhalten soll, wird explizit angegeben. Keine zentralen Speicherelemente wie Stacks oder Registersätze sind für Datenübertragungen zwischen Instruktionen nötig. Daher erhält die neue Architektur den Namen Data Flow Oriented Java Architecture (DOJA).

Im Rahmen dieser Arbeit werden sowohl grundlegende Prinzipien als auch wichtige Details von DOJA erklärt. Änderungen, die nötig sind, um den existierenden, auf Java Bytecode basierenden Prozessor an die neue Architektur anzupassen, werden beschrieben. Zudem wird die Software Toolchain vorgestellt mit Fokus auf speziellen Herausforderungen, die mit der neuen ISA auftreten. Programme können ausgehend von Assembly Code und Java Klassendateien generiert werden. Verschiedene Varianten von ISA und Hardwareimplementierung werden evaluiert, um eine optimale Konfiguration zu ermitteln. Der resultierende Prototyp wird mit dem auf Bytecode basierenden Prototypen verglichen unter Zuhilfenahme der SPEC JVM98 Benchmarks sowie einiger Mikrobenchmarks mit sehr flachen Aufrufhierarchien. Beide Prototypen enthalten identische ALUs und das gleiche Heap-Speichersystem. Beachtliche mittlere Beschleunigungen von 1,87 für die SPEC Benchmarks und 2,89 für die Mikrobenchmarks werden mit ähnlichen Hardwareressourcen erreicht. Die Mikrobenchmarks erhalten durch ein Coarse Grained Reconfigurable Array (CGRA) mit vier Rechenelementen eine zusätzliche Beschleunigung von 2,53 im Durchschnitt und von 8,97 im besten Fall.

Abstract

Java bytecode is widely used as binary program representation on many different hardware platforms. Inherent safety features like the strict memory model and runtime code checking make it interesting for embedded systems as well. Java processors, which execute Java bytecode directly on hardware without virtual machine and operating system layer, promise lower resource requirements and faster reaction to external events in this field of application. The bytecode-based AMIDAR processor is an example for such a Java processor. Adaptive Microinstruction Driven Architecture (AMIDAR) processors focus on runtime adaptivity. They map compute-intense code sections to reconfigurable hardware accelerators without user intervention. The high abstraction level of Java bytecode facilitates this mapping because the original intention of the programmer can be inferred more easily.

Java bytecode is a stack-based architecture. Data is transferred between instructions using a stack, which causes many unnecessary data transfers. The operand stack limits performance in many Java processors. This work solves the problem by developing a new instruction set architecture (ISA) as replacement for Java bytecode on AMIDAR processors. This ISA eliminates the operand stack while keeping the high abstraction level. Its design is data flow oriented. The instruction which shall receive the result of an operation is specified explicitly. No central memory element like a stack or a register file is required for data transfers between instructions. Therefore, the new architecture is named Data Flow Oriented Java Architecture (DOJA).

As part of this work, basic principles as well as important details of DOJA are explained. Changes required to adapt the existing bytecode-based processor prototype to the new architecture are described. Furthermore, the software tool chain is presented with focus on special challenges which arise with the new ISA. Programs can be generated from assembly code and Java class files. Different ISA and hardware implementation variants are evaluated in order to select an optimal configuration. The resulting prototype is compared to the bytecode-based prototype using SPEC JVM98 benchmarks and micro benchmarks with very flat call graphs. Both prototypes contain identical ALUs and the same heap memory system. Remarkable average speedups of 1.87 for SPEC benchmarks and 2.89 for micro benchmarks are achieved with similar hardware resources. Micro benchmarks gain further speedup of 2.53 on average and 8.97 at best by acceleration with a coarse grained reconfigurable array (CGRA) of four processing elements.

Contents

Part I.

Introduction

1. Introduction

Instruction set architectures (ISAs) are the key components of programmable computing hardware because they define the interface between hardware and software. They serve as starting point for optimizing both microarchitecture and compilers. Usually, development is focused on one of both. This can lead to inefficient mismatches, especially if application requirements or implementation technology change. After some time, changes to the ISA seem beneficial, which often leads to instruction set extensions. Introducing a complete new ISA happens rarely because of legacy software in binary form. Lately, a clear trend towards the RISC paradigm can be observed like Apple switching from x86 to ARM [1]. Processors based on the AMIDAR principle take a different approach [2]. They favor complex instructions operating on a higher abstraction level to facilitate dynamic software/hardware migration.

In this work, a new ISA for AMIDAR processors is introduced as replacement for Java bytecode. It speeds up execution of Java programs significantly while keeping the high abstraction level. An FPGA prototype of the new architecture shows that hardware resource consumption is kept almost equal.

1.1. Motivation

Motivation for this work is manifold. It can be condensed to the following four questions.

Why Java?

The basic concept of the Java platform is to compile source code to an abstract intermediate form named Java bytecode instead of compiling directly to target hardware and operating system [3]. Java bytecode is executed on a virtual machine. Only the implementation of this Java virtual machine (JVM) must depend on specific hardware and the operating system. This design increases portability tremendously. Today, Java bytecode is a universal format which is widely used across many devices. Besides the original Java language, many other programming languages like Scala, Groovy, or Kotlin can be compiled to Java bytecode. Strict object-oriented design as well as automatic memory management facilitate programming and increase productivity. Furthermore, the Java platform provides powerful constructs for multithreading. Last but not least, the feature-rich standard library and a huge amount of 3rd party libraries make the Java platform popular.

Why a Java processor?

The motivation for the first processors which used Java bytecode as their ISA was to gain performance compared to virtual machines. Early JVM implementations were based on interpretation. However, modern JVMs such as HotSpot [4] are based on just-in-time

compilation (JIT) instead. This speeds up execution enormously and thus renders the performance argument obsolete. On the other hand, safety features like runtime type checking and automatic memory management make the Java platform interesting for embedded systems as well. Java processors promise lower resource requirements and faster reaction to external events in this field of application because the operating system layer is eliminated.

Why AMIDAR?

The Adaptive Microinstruction Driven Architecture (AMIDAR) [5] is designed to simplify runtime adaptation to executed applications. It is achieved by fully separating a processor into functional units (FUs) which are largely independent from each other. This also simplifies hardware development due to higher modularity. Dynamic software/hardware migration is a special ability of AMIDAR. Compute-intense code sections are mapped to hardware accelerators without intervention by programmers. A high abstraction level of the ISA is beneficial for this process because the original intention of the programmer can be inferred more easily. Therefore, Java bytecode is a favorable choice as ISA for AMIDAR processors.

Why a new ISA?

Java bytecode uses a stack to transfer data between instructions. This indirection causes many unnecessary data transfers and slows down execution. Consequently, the stack usually becomes the bottleneck in bytecode processors [6]. Inefficiency is even higher for AMIDAR processors because of strict separation of FUs. Since many instructions both start popping operands from the stack and finish pushing results onto the stack, concurrent or overlapping execution of instructions is hampered severely. Sophisticated countermeasures like instruction folding have been introduced. However, they generate considerable overhead in terms of hardware resource consumption and length of the critical path [7]. Therefore, changing the ISA seems to be a rewarding method for diminishing these inefficiencies. Shifting to RISC architectures would make generic program execution more efficient but would also impede dynamic mapping to hardware accelerators. An ISA which keeps the high abstraction level of Java bytecode but avoids central memory elements like stacks or register files for transferring data between instructions would be the ideal solution. Because such an ISA does not exist yet, it is a promising research direction.

1.2. Research Contributions

This work provides the following research contributions:

- A new data flow oriented ISA is presented, which is optimized for AMIDAR processors. It is semantically compatible to Java bytecode so that Java programs can be executed natively. The ISA keeps the high abstraction level of Java bytecode but increases execution speed by removing unnecessary stack operations and data transfers.
- An assembly language for the new ISA is defined and a corresponding assembler program is implemented. A transpiler for converting Java class files into program images of the new ISA is engineered.

- A novel technique for matching operands with FU operations in AMIDAR processors is developed. It provides more concurrency and throughput than the previous technique based on tags.
- An FPGA prototype of the processor is implemented. It is capable of executing rather complex Java programs. Parameters of the ISA are optimized systematically with the help of behavioral HDL simulations of this prototype. A detailed comparison between Java bytecode and the new ISA is performed using the optimized prototype.

1.3. Thesis Outline

Part I, which contains this chapter, introduces into the research area of this thesis. Chapter 2 will explain technical fundamentals this work is based on. Subsequently, chapter 3 will give an overview of other research which has been conducted on hardware execution of Java bytecode or on data flow architectures.

Part II will illustrate the development process which has led to basic architecture design decisions. After analyzing weaknesses of the bytecode-based AMIDAR processor in chapter 4, chapter 5 will introduce basic design principles of the new ISA.

Part III will deal with implementation details. Chapter 6 will address several aspects of the ISA, whereas chapter 7 will cover all changes which are necessary to adapt the bytecode-based AMIDAR processor to the new ISA. Explanations of program image generation from assembly code and from Java class files will follow in chapters 8 and 9 respectively.

A thorough evaluation will be given in part IV. Basics of the evaluation process will be described in chapter 10. Afterwards, chapter 11 will evaluate several ISA and hardware implementation variants. The optimal configuration will be determined, which will be compared to Java bytecode in terms of performance, hardware requirements, and code size in chapter 12. Chapter 13 will assess automatic acceleration by CGRA context synthesis. Finally, chapter 14 will summarize this thesis, will discuss results, and will give an outlook on future research opportunities.

Some sections will contain literal excerpts from the own publications [8] and [9]. Their source will be stated but they will not be marked as literal citations explicitly.

2. Technical Background

This chapter introduces the fundamental techniques this work is based on. First, various aspects of the Java platform are explained. Then, the AMIDAR concept is presented together with important details of the current processor implementation, which uses Java bytecode as ISA. Finally, the CGRA which is applied as reconfigurable hardware accelerator in this processor is covered.

2.1. Java

The Java platform has been released by Sun Microsystems in 1995. Originally, it has been designed for delivering applications to consumers over the internet. Consequently, hardware platform independence and a compact program representation are two of its key features. Hardware platform independence is achieved by executing programs on virtual machines [3]. The corresponding architecture of the Java platform on general purpose computers is shown in fig. 2.1. An application is running on top of a comprehensive class library. Both are stored in form of class files which are the binaries to be loaded and executed by a Java virtual machine (JVM). Class files normally are independent of hardware architecture and operating system. All target specific components are part of the JVM, which relies on the underlying operating system and hardware. The Java Native Interface (JNI) [10] provides the opportunity to bypass the JVM and call functions implemented in native code. However, this makes the application platform specific again. It is required only if special functions of operating system or hardware must be accessed. Platform independence is one reason why Java is widely used today across many different devices from smart cards over blu-ray players to desktop computers and servers. However, different Java runtime specifications [11, 12] exist for these devices, which adapt the range of functions to device capabilities.

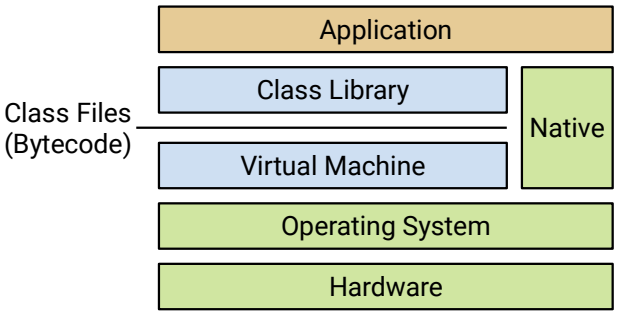


Figure 2.1.: Architecture of the Java platform on general purpose computers

The Java programming language [13] is the third core component of the Java platform besides JVM and class library. A Java compiler translates Java source files into class files as

depicted in fig. 2.2. It typically produces unoptimized code. Optimization opportunities are left for the JVM to better exploit target specific features. Many alternative programming languages which can be compiled to class files have been developed like Groovy, Scala, Clojure, and Kotlin. Consequently, Java class files have evolved into a universal program representation. The Java platform is used partly to develop apps for Android smartphones as well. Instead of running class files on a virtual machine, they are converted into Dalvik executables, which are compiled into native code during installation [14].

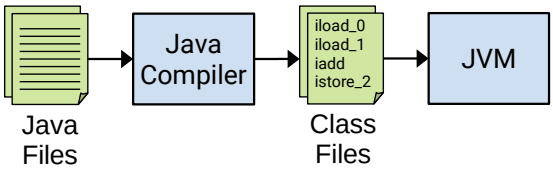


Figure 2.2.: Tool flow for running programs on the Java virtual machine

2.1.1. Programming Language

The Java programming language [13] has a syntax similar to C/C++ but is more restrictive in some aspects. The programmer's freedom is reduced systematically to avoid common mistakes and to encourage object-oriented, modular software design. For example, every function or variable must be part of a class. There are no global functions or variables. All variables are typed statically with a clear separation between primitive types like numbers and complex types like objects or arrays. Primitive types are always allocated on stack and are copied by value. Complex types are always allocated on heap and are copied by reference. No pointer arithmetic is available. Memory allocated on heap does not need to be released manually. The JVM detects which objects cannot be reached by references any more and deletes them automatically. This technique is called *garbage collection*.

Each class can inherit method implementations or fields only from one direct super class. Interfaces are special classes which can declare only abstract methods without implementations or static methods. Classes can implement multiple interfaces, which means inheriting their abstract method declarations. All classes and arrays inherit from `java.lang.Object` by default. Therefore, "objects" will also include arrays if not stated differently from here on.

The language provides multithreading features inherently. Every object can serve as *monitor*, which is an extended mutex variant. Code blocks or methods can be specified as *synchronized* to a monitor. Then, only the thread which holds the monitor can enter this synchronized block. Because a monitor can be held by at most one thread, this ensures that only a single thread can be executing each synchronized block at any point in time. Furthermore, monitors provide a simple mechanism for sending signals between threads. If a thread holds a monitor, it can wait for a signal on this monitor. It continues executing only after another thread has sent a notification signal to the monitor.

Listing 1 shows a very simple Java program. The main method of this program will be used as running example throughout this thesis. The program calculates the scalar product of two arrays and prints the result to the standard output. Arrays are initialized statically and are stored in static fields for the sake of simplicity. The loop iterates over the elements in backward direction to reduce the number of instructions in the loop body as generated by the Java compiler.

```java
package app;

public class RunningExample {

    private static int[] arrayA = {100,200,300,400,500,600,700,800,900};
    private static int[] arrayB = {100,200,300,400,500,600,700,800,900};

    public static void main(String[] args) {
        int[] a = arrayA;
        int[] b = arrayB;
        int sum = 0;

        for (int i = a.length - 1; i >= 0; i--) {
            sum += a[i] * b[i];
        }

        System.out.println(sum);
    }
}
```

Listing 1: Java source code for the running example

2.1.2. Virtual Machine

The JVM [3] executes Java programs in form of class files. The full object-oriented nature of Java source code is preserved in these files. Each file holds information about one class, which consists of the following parts:

- The constant pool stores all constants (mainly numbers or strings) which are used in the rest of the file.
- Java bytecode encodes method bodies. Classes, fields, or methods are referenced by name strings which are stored in the constant pool.
- Additional meta information comprises the super class, implemented interfaces, fields, methods, and further attributes. Name strings and type information are provided. This information is used for dynamic linking and runtime type checking by the JVM. Furthermore, it can be exploited for compiling against the class. Hence, header files like in C/C++ are not required.

Java bytecode is of major importance for this work because it serves as ISA for the JVM. It makes use of three data areas: operand stack, local variables, and heap. The operand stack serves as implicit source and sink of data for most instructions. It is the primary means for transferring data between instructions. For example, the `IADD` instruction pops two operands from the operand stack, adds them as integer values, and pushes the result back onto the stack. Local variable (LV) storage holds the values of method local variables. It can be compared to stack memory in other RISC or CISC architectures. Both operand stack and LV storage for one method are combined to a method frame. Method frames are organized as stacks. A new frame is pushed on method invocations and the top frame is popped on method returns. It is important to distinguish between these stacks of method frames, which exist once per thread, and the operand stacks, which exist once per method

invocation. Operand stack and LVs can store only primitive values or references to objects. Objects and static class fields are stored in heap memory. All objects are annotated with type information, which enables runtime type checking.

Java bytecode instructions are structured in bytes and have variable length. The first byte is the operation code. It is usually followed by 0 to 4 bytes, which are treated as parameters. Switch instructions have much more parameter bytes for encoding jump targets. Instructions can be categorized as follows:

- Load/store LVs or constants (`ILOAD`, `FSTORE`, `LDC`, ...).
- Execute arithmetic/logic operations (`IADD`, `FSUB`, `LAND`, ...).
- Convert primitive types (`I2L`, `L2F`, `D2I`, ...).
- Create or manipulate objects (`NEW`, `GETFIELD`, `IASTORE`, ...).
- Manipulate the operand stack directly (`POP`, `DUP`, `SWAP`, ...).
- Influence control flow (`GOTO`, `IFEQ`, `LOOKUPSWITCH`, ...).
- Change methods (`INVOKESTATIC`, `RETURN`, `IRETURN`, ...).
- Throw exceptions (`ATHROW`).
- Synchronize threads using monitors (`MONITORENTER`, `MONITOREXIT`).

Methods, fields, and classes are referenced by indices into the constant pool. The indices point to data structures which in turn point to strings. They define names and types of the desired entities. This allows dynamic linking of classes. Five types of method calls exist:

- Virtual calls invoke polymorphic object methods by referencing classes. The target method of a virtual call depends on the runtime class of the object on which the method is invoked. Thus, the target is determined dynamically at runtime. This process is called *dynamic resolution*. Because each class can have only a single parent class, such calls can be realized by simple lookups in virtual method tables.

- Interface calls invoke polymorphic object methods by referencing interfaces. Because each class can implement multiple interfaces, dynamic resolution of interface calls is more complex than resolution of virtual calls.

- Static calls invoke static methods directly. Dynamic resolution is not necessary.

- Special calls invoke object methods directly without dynamic resolution. They are applied for constructors and private methods.

- Dynamic calls invoke bootstrap methods, which construct call targets dynamically, on first invocation. These call targets will be used for subsequent invocations. Dynamic calls realize lambda expressions efficiently when compiling from Java source code.

Resulting bytecode for the main method of the running example is depicted in fig. 2.3. References into the constant pool are replaced by corresponding strings for clarity. Line 1 loads the reference to the array stored in static field `arrayA` onto the operand stack. Line 2 pops this value and stores it into LV slot 1 (variable `a`). Lines 3 and 4 store the reference to `arrayB` into slot 2 (variable `b`). Lines 5 and 6 store the integer constant 0 into slot 3 (variable `sum`). Lines 8 to 12 load the reference to array `a` from slot 1, obtain its length, subtract the integer constant 1, and store the result into slot 4 (variable `i`). Lines 14 and 15 check the loop condition by comparing `i` to zero. Lines 17 to 26 compute the next value

of `sum`. This is done by loading all required variables onto the operand stack, reading the array elements (lines 20 and 23), executing the arithmetic operations, and storing the result into slot 3. All intermediate values are transferred over the operand stack here. Line 28 decrements `i` by 1. This is the only arithmetic operation of Java bytecode which can be executed directly on LVs without using the operand stack. Line 29 jumps back to the beginning of the loop. Line 31 pushes the reference to the `PrintStream` object of the standard output onto the stack. It is used as first implicit argument to the `println(int)` method. The second argument is loaded by line 32. Line 33 calls the `println(int)` method virtually. After this method is finished, the main method returns with line 35.

```
1   GETSTATIC app/RunningExample/arrayA
2   ASTORE_1
3   GETSTATIC app/RunningExample/arrayB
4   ASTORE_2
5   ICONST_0
6   ISTORE_3
7
8   ALOAD_1
9   ARRAYLENGTH
10  ICONST_1
11  ISUB
12  ISTORE 4
13 L1:
14  ILOAD 4
15  IFLT L2
16
17  ILOAD_3
18  ALOAD_1
19  ILOAD 4
20  IALOAD
21  ALOAD_2
22  ILOAD 4
23  IALOAD
24  IMUL
25  IADD
26  ISTORE_3
27
28  IINC 4 -1
29  GOTO L1
30 L2:
31  GETSTATIC java/lang/System/out
32  ILOAD 3
33  INVOKEVIRTUAL java/io/PrintStream/println(I)V
34
35  RETURN
```

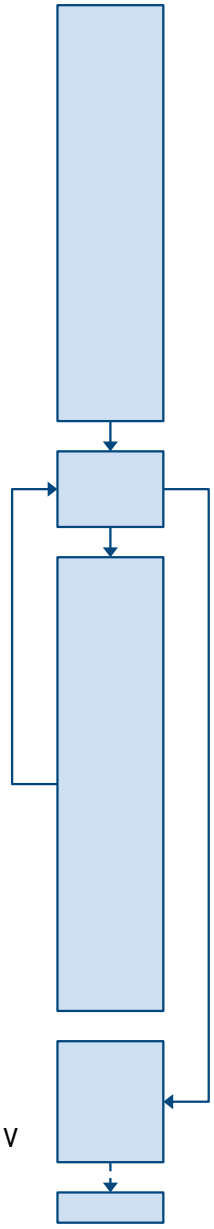


Figure 2.3.: Java bytecode for the running example with illustration of control flow

Exception handling, which is not shown in the above example, is realized using exception handlers. Each method can specify an ordered list of exception handlers. A handler defines the range of instructions for which it handles exceptions, the type of handled exceptions,

and the code address which should be executed for handling exceptions. If an exception is thrown, the JVM searches for the first matching handler of the current method and continues execution at the corresponding code position. If no handler is found, the frame of the current method is popped and the exception is rethrown in the context of the invoking method. If none of the methods on the call stack specify a matching handler, the thread is terminated.

The JVM specification does not define by which technique bytecode is executed. The first implementations used interpretation while most modern implementations are based on just-in-time compilation (JIT) to increase performance. For example, HotSpot [4] starts executing by interpretation. Compute intense parts of the code are compiled at runtime. Compiler optimizations are guided by runtime profiles of the program. Garbage collection is another feature which is not specified in detail. Typically, three steps are necessary:

1. The root set of reachable objects must be collected. It contains all objects which can be accessed directly by any thread of the program. Therefore, all existing method frames (LVs and operand stacks) must be examined for object references. In addition, references stored in static fields contribute to this set.

2. All other objects which can be reached indirectly through object fields or array elements must be traced.

3. Objects which have been identified to be unreachable are deleted. Heap memory is compacted if necessary.

2.2. AMIDAR

Java bytecode processors execute bytecode directly on hardware without JVMs. Figure 2.4 shows the architecture of the Java platform on the AMIDAR-based bytecode processor. This depiction is valid for Java bytecode processors in general. The virtual machine and operating system layers have been removed compared to fig. 2.1. The interface to native code is also missing because Java bytecode is the native code for Java processors. Consequently, functions which have been undertaken by the JVM or the operating system must now be implemented as part of hardware or class library.

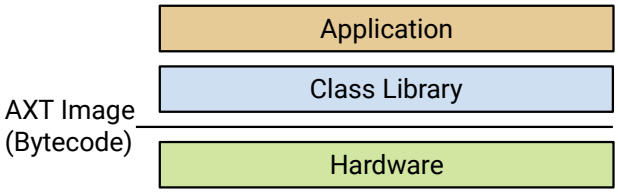


Figure 2.4.: Architecture of the Java platform on AMIDAR

The primary idea of the Adaptive Microinstruction Driven Architecture (AMIDAR) [5] is to fully separate a processor core into FUs which are largely independent from each other. This facilitates hardware design and self-optimization by runtime reconfiguration. The AMIDAR principle is not specific for Java processors but can be applied to any ISA. The hardware architecture of the full AMIDAR system [15, 16], which is implemented on a

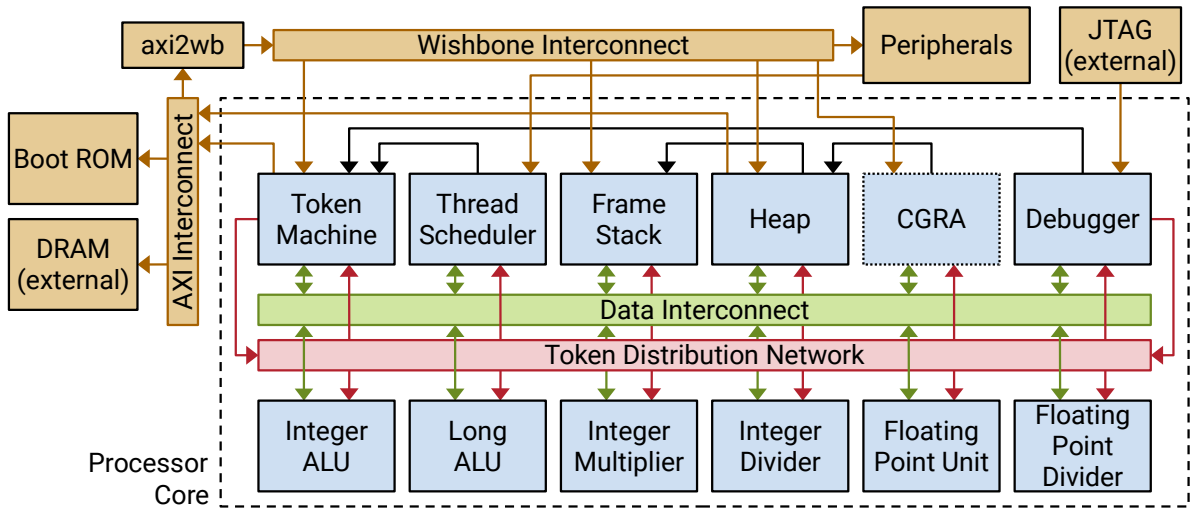


Figure 2.5.: Hardware architecture of the bytecode-based AMIDAR system

Xilinx Artix 7 FPGA, is illustrated in fig. 2.5. Arrows point from masters to slaves in this picture.

The Token Machine is the most important FU. It reads bytecode instructions and converts them into sets of tokens, which are sent to all FUs over the token distribution network (TDN). A token contains the operation the FU should execute and the target of the operation result. Conversion from bytecode into tokens as well as execution of tokens by FUs will be discussed in more detail in section 4.1. Data is exchanged between FUs using a common data interconnect. When implementing an FU, assumptions about the timing of other FUs, of the data interconnect, or of the token distribution network are neither allowed nor necessary. Communication between FUs is self-timed. [9] The individual FUs have the following functionality:

- **Token Machine (TM)**: Decodes all instructions and executes tokens which change control flow like branches or invocations. Furthermore, it provides constants to the system and executes inheritance checks. Instructions as well as required meta information are fetched from external memory via AXI interfaces to internal caches.
- **Frame Stack (FS)**: Provides one stack of method frames for each live thread. Method frames hold LVs and operand stacks.
- **Heap**: Stores objects and static fields in external memory via AXI interface. It contains the data cache system. Each field is addressed using handle and offset. The handle references the object and the offset addresses the field within the object. The mapping from handles to physical addresses is stored in the handle table, which is located in external memory and is cached inside the FU. All static fields are stored in a special object with handle 1. Garbage collection is also implemented here. A dedicated connection from Heap to Frame Stack is required for collecting the root set of reachable objects. The Heap FU provides access to peripherals by mapping special peripheral objects to corresponding physical addresses.
- **Thread Scheduler (TS)**: Decides which thread should be executed and implements thread synchronization by monitors. Thread switch requests are sent to the Token

Machine over a dedicated connection. The Thread Scheduler handles interrupt requests from peripherals by switching to associated interrupt service threads (ISTs).

- **CGRA**: Reconfigurable hardware accelerator which is explained in section 2.3. A dedicated connection provides access to the data cache system of the Heap. The CGRA is an optional FU.
- **Debugger (DBG)**: Can act as second Token Machine which is controlled by an external development system over a JTAG interface [17]. It can start and stop normal program execution using a dedicated connection to the Token Machine.
- **Integer ALU (IALU)**: Executes arithmetic and logic operations on 32 bit integer operands excluding multiplication and division.
- **Long ALU (LALU)**: Executes arithmetic and logic operations on 64 bit integer operands excluding multiplication and division.
- **Integer Multiplier (IMUL)**: Multiplies 32 bit or 64 bit integer operands.
- **Integer Divider (IDIV)**: Divides 32 bit or 64 bit integer operands.
- **Floating Point Unit (FPU)**: Executes arithmetic operations on single or double precision floating point operands excluding division.
- **Floating Point Divider (FDIV)**: Divides single or double precision floating point operands.

Token Machine and Heap can access memories and peripherals over an AXI interconnect [18]. An external DRAM is used for storing the program image and heap data. The boot ROM contains a boot loader which receives the program image over UART and initializes FUs. AXI interconnect, DRAM controller, and boot ROM are realized by Xilinx IP cores. A part of the AXI address range is reserved for peripherals. Requests in this range are converted to Wishbone [19]. The Wishbone slave interfaces of Token Machine, Frame Stack, and Heap are mainly used for configuration by the boot loader and to obtain status information. The Wishbone slave interface of the CGRA is used for writing to configuration memories. Processor core and AXI interconnect run at 100 MHz. Wishbone interconnect and peripherals run at a phase-aligned 25 MHz clock.

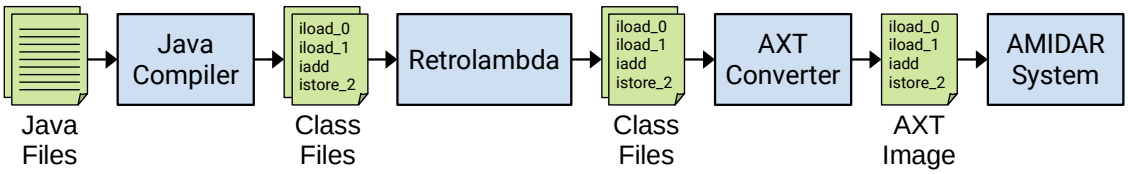


Figure 2.6.: Tool flow for running programs on bytecode-based AMIDAR systems

Although Java class files could be used as binaries for Java processors, AMIDAR combines all class files into a single program image, which is called AMIDAR executable (AXT) [15]. The corresponding tool flow is shown in fig. 2.6. After compilation into class files, two additional steps are necessary. Class files are passed through the Retrolambda tool [20]. It converts all dynamic invocations which originate from source code lambda expressions

into methods which can be invoked through interfaces. This step is required because AMIDAR does not support dynamic invocations natively. Afterwards, resulting class files are converted into an AXT image which can be executed by the processor. The AXT converter statically links class files by replacing all symbolic references by fixed addresses. Furthermore, it converts class meta information into a form which can be evaluated by hardware more easily.

2.2.1. Data Interconnect

The data interconnect and the mechanism for matching data with FU operations is of special interest for this work. Therefore, it is explained in more detail here. This explanation has been published previously in [9]. The processor uses 64 bit wide dedicated data connections between FUs. Only the connections required by the token sets are realized. As most token sets use the operand stack as source and sink of data, this effectively results in a star topology with the Frame Stack as its center.

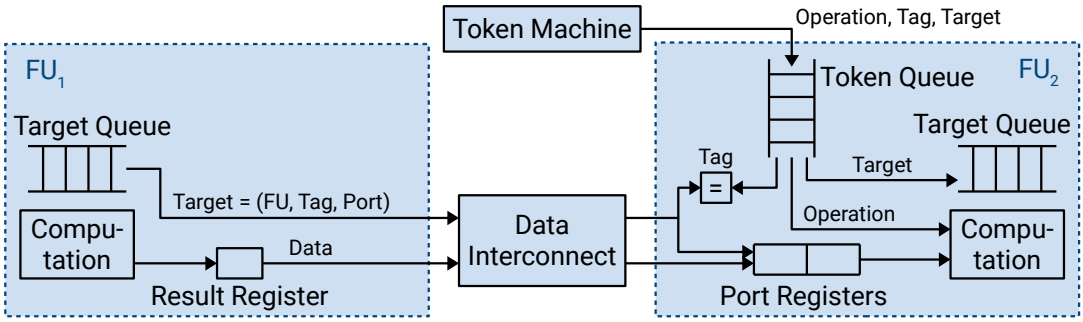


Figure 2.7.: Data transmission in the bytecode-based AMIDAR system [9]

Tags are used for matching data with FU operations. The required hardware components are depicted in fig. 2.7. The token machine puts operation, tag, and target into the *token queue* of an FU. A target consists of the FU where the result should be delivered to, the tag of the operation which uses the result as operand, and a port which allows to distinguish between multiple operands of one operation. When all operands of the operation at the front of the token queue have been received in the *port registers*, the operation is executed and its target is stored into the *target queue*. When computation is finished, the result is stored into the *result register*. The result is sent to the target FU using information from the target queue. Data is accepted and stored in a port register only if its tag matches the tag of the operation at the front of the token queue. Otherwise, the source FU must retry until data is accepted. It must be noted that target queues exist only from logical point of view. FUs implement them as pipeline registers which carry target information along with data during computation.

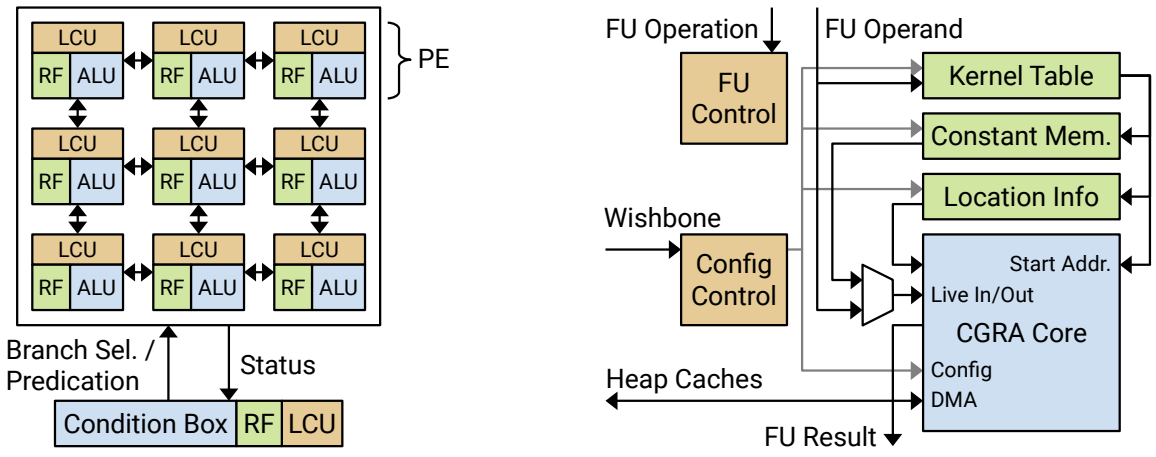
Data transmission requires one clock cycle from result register to port register. The sender receives the acknowledgment one additional clock cycle later. One transmission is allowed per port register and clock cycle. Each incoming data connection from another FU has its own tag comparator. The request with matching tag is accepted. Since there can be only one request with matching tag, no further arbitration is required by the data interconnect. The weaknesses of this matching technique will be analyzed in section 4.1.

2.2.2. Garbage Collection

The Heap FU implements semi-concurrent garbage collection in hardware. Heap memory space is divided into two sections. If the number of free handles runs below a certain threshold or a new object does not fit into the active section, normal heap operation is blocked and garbage collection is triggered. The root set of reachable objects is collected and all reachable objects are traced. The active section is switched and normal heap operation can continue. While new objects are being allocated in the active section, the garbage collector compacts the inactive section concurrently by copying all reachable objects to the lowest possible memory addresses. The memory addresses of individual objects can be locked in order to allow direct memory access (DMA) for peripherals. The garbage collector skips copying these objects during the compaction phase.

2.3. CGRA

Coarse grained reconfigurable arrays (CGRAs) [16] are computing architectures which provide programmable data path primitives. Unlike FPGAs, they are not programmable on bit level but on word level. On the one hand, this makes them less flexible than FPGAs. On the other hand, configuration complexity is reduced, which allows CGRA configurations to be synthesized much faster. Furthermore, configuration size is decreased, which enables storing multiple configurations and switching them every clock cycle. One such configuration is typically called *context*. Consequently, CGRAs can be used as hardware accelerators which are (re)configured at runtime. AMIDAR integrates such a CGRA tightly into a processor core. It identifies compute-intense parts of the program, called *kernels* from here on, and synthesizes them into CGRA contexts. The CGRA variant presented in this section is integrated into the processor which is developed along with this thesis. It is suitable for accelerating other processor architectures as well [21]. The bytecode-based processor uses an older variant instead which differs in details of the CGRA core and the synthesis process.



(a) CGRA core (b) CGRA FU

Figure 2.8.: Hardware architecture of the CGRA FU

The design of a CGRA core is depicted in fig. 2.8a. It consists mainly of an array of processing elements (PEs) which have limited data connections among each other. Each

PE comprises a register file (RF), an ALU, and a local configuration unit (LCU). The LCU contains a context memory and a context counter, which provide PE configurations. Register file sizes and supported ALU operations can vary between PEs. Furthermore, some PEs can have DMA connections. Additionally, each CGRA core has one condition box, which receives status signals from PEs. It generates branch selection signals for the LCUs of the system as well as predication signals for register files and DMA. The condition box has an own register file and an own LCU, which defines how control signals are generated.

AMIDAR processors integrate CGRA cores as FUs because this enables fast transfer of method-local data to and from the CGRA. Data which must be sent to the CGRA before executing a kernel is called *live-in* data. Conversely, data which must be retrieved from the CGRA after executing a kernel is called *live-out* data. The architecture of the CGRA FU is shown in fig. 2.8b. Three memories are contained in this FU besides the CGRA core. The kernel table has one entry for each kernel which has been programmed into the CGRA. An entry is composed as follows:

- The starting address of kernel contexts in context memories of the core.
- The starting address of kernel constants in constant memory.
- The number of kernel constants.
- The starting address of location information entries for the kernel.

The constant memory stores all kernel constants. The location information memory stores CGRA core addresses of live-in/out values and constants. Each such address consists of PE number and register file address. These memories as well as all context memories of the core are filled via the Wishbone interface of the FU. Afterwards, executing a kernel requires the following steps:

1. The `INIT` operation is issued with the kernel ID as operand. This ID selects an entry of the kernel table. The addresses contained in this entry are passed to constant memory, location information memory, and CGRA core. All constants of the kernel are transferred from constant memory to the register files of the CGRA core. The destination addresses for the constants are taken from location information memory. The location information memory address is incremented for each constant.

2. `PUSH` operations are used for transferring live-in values from the Frame Stack to the register files of the CGRA core. Destination addresses are taken from location information memory again.

3. The `RUN` operation starts kernel execution. Afterwards, the FU sends the constant 0 as result of this operation. This constant allows to synchronize normal program execution with the CGRA core if there are no live-out values.

4. `PULL` operations are used for transferring live-out values from the register files of the CGRA core to the Frame Stack. Source addresses are taken from location information memory.

This process is triggered by special bytecode instructions which are patched into the program code of the kernel start. These instructions activate the micro-coded generation of tokens with the operations mentioned above. Afterwards, the instructions jump to the first instruction after the kernel where normal program execution continues.

Each PE which provides DMA for reading and modifying objects is linked to an own L1 data cache inside the Heap FU. These caches are connected to a common L2 data cache. Coherence between L1 caches is ensured by the Dragon protocol [22].

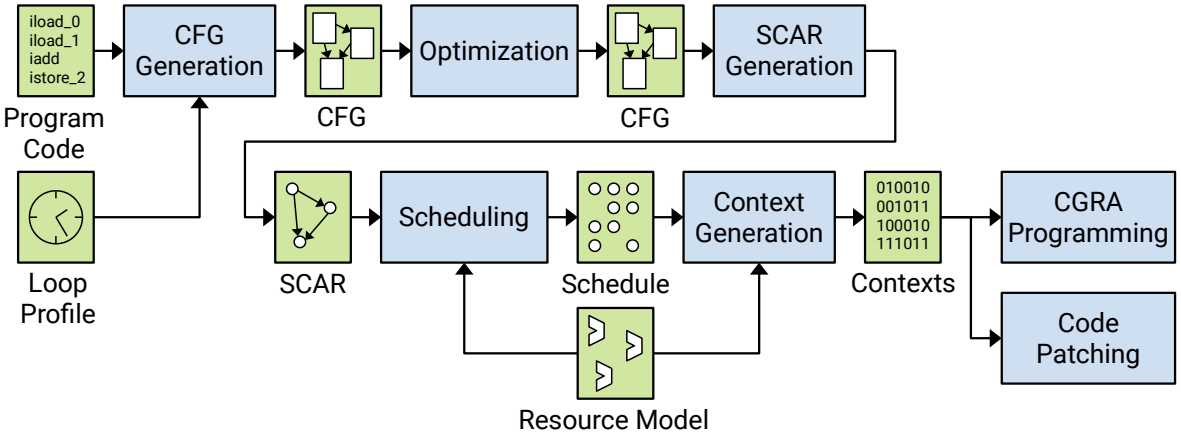


Figure 2.9.: Tool flow for synthesizing CGRA contexts from a program loop

Figure 2.9 illustrates the tool flow for generating CGRA contexts from loops of the program code. A profiler, which is integrated into the Token Machine, detects loops and measures how many instructions are executed by these loops. The synthesis thread periodically checks for loops which are suitable for acceleration. Suitable loops account for at least 30 % of executed instructions since the last check. The loop with highest instruction count is selected for synthesis. Its program code is read from code memory and is converted into a control flow graph (CFG) representation which is independent from both the processor architecture and the concrete CGRA architecture. This representation is very similar to static single assignment (SSA) forms often found in compiler engineering. Several optimizations are executed on CFGs like method inlining and loop unrolling. Afterwards, the CFG is converted into the scheduler application representation (SCAR). It defines explicit control and data dependencies between PE operations. SSA form is eliminated during this step. Then, SCAR is scheduled onto the resource model of the CGRA core. Binary contexts to be loaded into context memories of the CGRA are generated from the schedule. Finally, the synthesis thread waits until the loop is not being executed. Then, it patches the program code to trigger CGRA execution for future invocations of the loop.

The CGRA schedule which results from the loop of the running example (listing 1) with unroll factor 2 is depicted in fig. 2.10. The corresponding PE topology is given in fig. 2.11. Register names are composed of PE number before the colon and local register address after the colon. “C” indicates registers of the condition box. Each PE has a status output “n:S”. The condition box has a branch selection output “C:B”. Registers are assigned sequentially whereas status and branch selection outputs are assigned combinationally. Table 2.1 defines the register contents before (live-in) and after (live-out) executing the kernel.

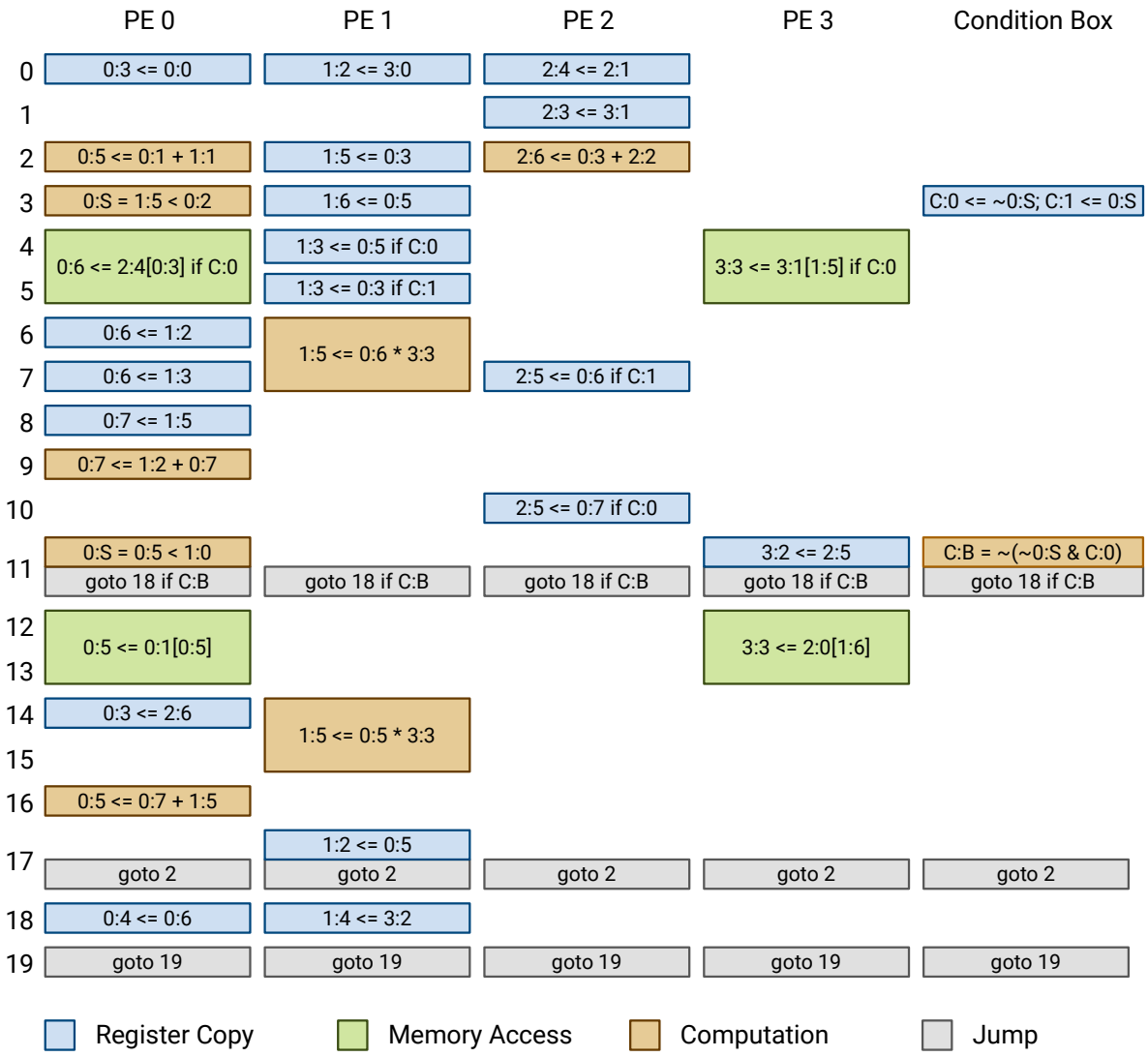


Figure 2.10.: CGRA schedule for the loop of the running example

Figure 2.11.: PE topology used for scheduling the running example

Live-In Register	Content	Live-Out Register	Content
0:0	i	0:4	i
0:1	a	1:4	sum
0:2	0	2:3	b
1:0	0	2:4	a
1:1	-1		
2:0	b		
2:1	a		
2:2	-2		
3:0	sum		
3:1	b		

Table 2.1.: Live-in and live-out values of the exemplary CGRA kernel

3. Related Work

Many other Java bytecode processors exist besides AMIDAR. Comparing them discloses strengths and weaknesses, which in turn give ideas for improvements. Therefore, study of related work starts with an overview of pure Java processors and how execution of Java bytecode can be accelerated in these processors. Techniques which accelerate the execution of Java bytecode by extending general-purpose processors exist as well [23, 24, 25] but are not presented here in detail.

It turns out that modifying the ISA might be beneficial. Consequently, other computing architectures than the JVM are examined. As the field of computing architectures is vast, a focus is required. AMIDAR is composed of independent FUs which apply data-driven synchronization among themselves. Hence, data flow architectures are a promising source of ideas. In addition, transport triggered architectures (TTAs) are studied because their structure has similarities to AMIDAR.

3.1. Java Processors

Although Java bytecode has originally been designed to be interpreted by a virtual machine, the first processor for executing bytecode natively, picoJava-I, has been proposed shortly after the public release of the Java platform [6, 26]. In the following years, many Java processors have been developed to speed up execution of Java bytecode by eliminating the virtual machine and operating system layers. Some of them are presented in the following sections.

3.1.1. picoJava-II

picoJava-I and its successor picoJava-II [27, 28] have been developed by Sun Microsystems. Sun has never produced chips for these designs but has published them under an open-source license finally. One challenge of implementing a bytecode processor is the great variance in instruction complexity. As many other Java processors, picoJava-II solves this problem by a RISC pipeline for simple instructions, which is assisted by microcode for multi-cycle instructions. The most complex instructions for memory management, thread synchronization, or exception handling trigger software traps. Memory management is accelerated by hardware support for handles and write barriers. Additionally, two hardware lock counters exist to speed up monitor locking. The processor implements folding of up to four instructions to accelerate generic bytecode execution. This technique will be explained in more detail in section 3.2.1. Another widely adopted technique for faster execution is runtime replacement of instructions which have symbolic references. These instructions originally reference methods or fields by name strings (see section 2.1.2). Such instructions are executed in software and are replaced by “quick” versions, which use the resolved addresses instead of symbolic references.

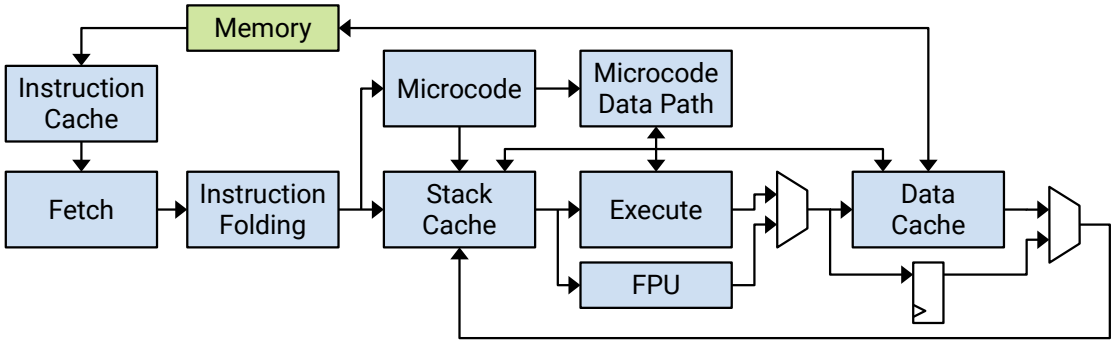


Figure 3.1.: Architecture of picoJava-II (adapted from [27])

The 6-stage execution pipeline is depicted in fig. 3.1. The first stage fetches bytecode from an instruction cache. The next stage decodes and folds it. The third stage consists mainly of a register file with 64 entries of 32 bits. This register file serves as cache for the operand stack and LVs. Its contents are spilled and filled via the data cache. The next stage executes arithmetic operations. A microcode driven floating point unit is available besides the integer data path. The fifth stage writes or reads memory contents using a data cache. The final stage writes data back into the register file. Multi-cycle instructions trigger the microcode sequencer, which can make use of an extended data path.

3.1.2. Jamuth

Komodo [29] and its successor Jamuth [30] are Java processors with low hardware requirements which explicitly target hard real-time applications. Jamuth contains four parallel instruction fetch stages and a segmented operand stack cache. Segments can be assigned exclusively to real-time threads to avoid swapping. Furthermore, a thread scheduler is implemented in hardware. These features allow zero-cycle thread switches. Even short latencies caused by branches or memory accesses can be utilized by switching to another thread. So-called *interrupt service threads* handle external events instead of interrupt service routines. One hardware thread slot can be shared by multiple non-real-time threads.

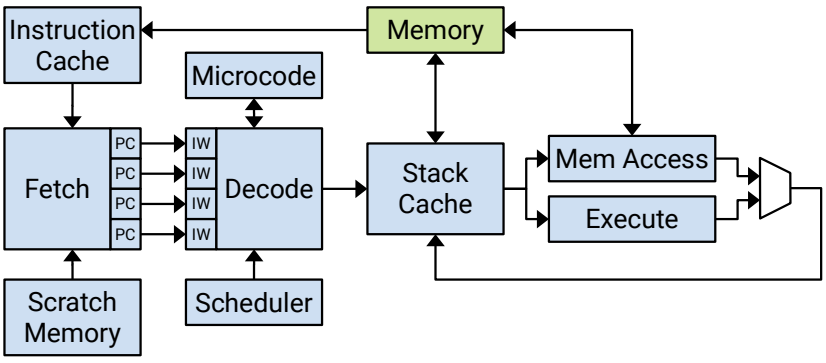


Figure 3.2.: Architecture of Jamuth (adapted from [30])

The computing architecture is sketched in fig. 3.2. A scratch memory can be used to store instructions besides the instruction cache. This memory has predictable and faster

access times, which make it ideal for real-time threads or trap routines. The fetch stage has four program counters (PCs) to fetch instructions for four threads in parallel. Fetched instructions are stored in instruction windows (IWs), which are small FIFO buffers. The hardware scheduler decides which of these windows is decoded. Each bytecode can either be decoded directly into a hardware instruction, or can trigger a microcode sequence, or can trigger a trap routine. The next stage reads operands from the stack cache. The fourth stage executes arithmetic operations or memory accesses. No data cache is used in order to avoid unpredictable timings. The final stage writes data to the stack cache.

3.1.3. JOP

Another processor designed for real-time systems is JOP [31, 32]. It is focused on a simple and time-predictable architecture. Therefore, the processing pipeline is simplified in comparison to previous architectures. All bytecode instructions are translated into microcode sequences, which are executed in the following stages. The sequence of a simple bytecode instruction might consist of one microinstruction only. This avoids having to distinguish between different types of instructions.

The runtime system is much more restrictive than normal real-time standards for the Java platform in order to allow exact worst-case execution time predictions. An application starts by calling all static initializers. Then, the initialization phase is executed. Global objects are allocated and threads are created in this phase. Finally, the mission phase begins with support for real-time constraints. Neither may new threads be created nor may their priorities change from this time on. Additionally, the wait/notify mechanism is forbidden. Threads may either execute periodic tasks or handle sporadic events.

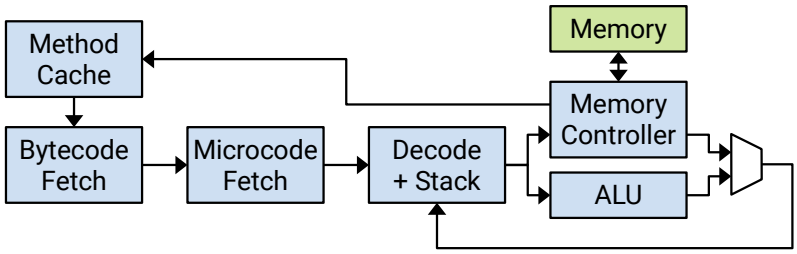


Figure 3.3.: Architecture of JOP (adapted from [32])

The 4-stage pipeline is shown in fig. 3.3. The concept of a method cache has been introduced. This memory stores the code of one whole method. It is filled on method returns, invocations, or thread switches. This makes delays caused by transferring instructions from the external memory to the cache much more predictable. The first stage fetches bytecode instructions from this cache and maps them to jump addresses for the microcode ROM. Bytecode jumps are executed in this stage as well. The next stage reads microinstruction sequences from the ROM and executes microcode jumps. The third stage decodes microinstructions and reads data from the stack. The final stage executes arithmetic operations or memory accesses and writes results back into the stack. Neither an extra write-back phase nor data forwarding are required because of a careful stack design. The stack stores all operands and LVs in on-chip memory. Two additional registers provide fast access to the values at the top of the stack. The stack memory holds additional

LV slots to be leveraged by microcode sequences. A hardware multiplier can be added as an optional extension.

3.1.4. SHAP

SHAP [33] also provides predictable execution times for hard real-time constraints. On the other hand, it supports advanced non-real-time features like dynamic class loading. Its structure as illustrated in fig. 3.4 has many similarities to JOP. A method cache is used as more predictable alternative to an instruction cache. Furthermore, bytecode is executed by a 4-stage pipeline with translation to microcode. An on-chip memory stores operand stacks and LVs. The two top stack elements are stored in registers. The major difference to JOP in terms of pipeline design is the improved stack management. SHAP allows dynamic allocation of method stacks for new threads. This is realized by storing each stack as linked list of blocks. Hard-coded operations are provided for managing method frames on invocations or returns. Furthermore, a separate data memory with 128 entries exists to be used in microcode sequences. It stores variables, status information, and constants.

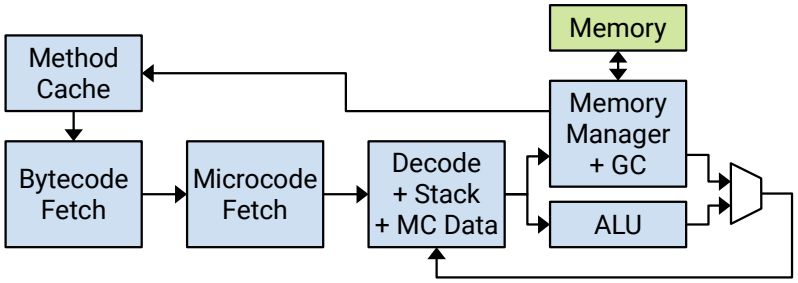


Figure 3.4.: Architecture of SHAP (adapted from [33])

Heap memory management has been extended by an autonomous garbage collector implemented in hardware. It runs concurrently without occupying the normal processing pipeline. Additionally, the memory manager stores the table which maps object references to physical memory addresses in hardware. This speeds up heap accesses but limits the number of reachable objects. The ISA has been modified slightly. For example, branch offsets are relative to method start, native methods are implemented by additional instructions, and special instructions for loading string constants have been introduced. The linker modifies the code accordingly when a program image is created.

3.1.5. aJ-200

aJ-200 [34] is a commercial microcontroller produced as ASIC. It contains a JEM2 Java processor core, which allows to execute two programs concurrently with full memory protection. Scheduling and synchronization is implemented in microcode, which leads to fast context switches in the region of 1 μs. A part of the microcode memory is programmable. Hence, custom bytecode extensions can be implemented. The basic structure of a JEM2 core is shown in fig. 3.5. No details about the pipeline structure are provided by the reference manual. A combined cache for instructions and data is used. Fetched bytecode is translated into microinstructions. A register file stores important pointers and the 6 top

elements of the stack. The datapath provides a separate integer multiply-accumulate unit and a floating point unit besides the 32 bit ALU.

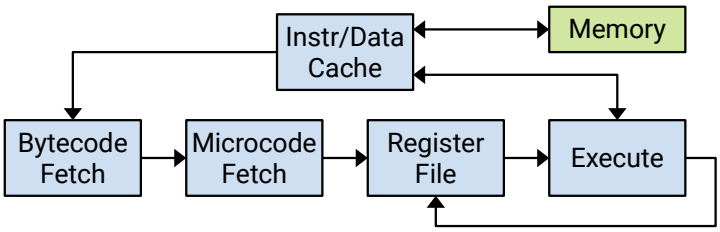


Figure 3.5.: Architecture of aJ-200 (adapted from [34])

3.1.6. JAIP

One of the latest Java processors presented in literature is JAIP [35, 36]. Originally, it has been designed as co-processor to work together with a RISC core. Finally, the RISC core has been removed. All required functionality has been implemented inside the Java processor or as Java software running on top of the processor. One unique feature is the full support for dynamic class loading. The power-on boot logic initializes the class loader from an internal memory. Then, the class loader is executed on the processor and loads classes from a standard Java archive (JAR) file on an SD card whenever they are required. No special program image must be prepared.

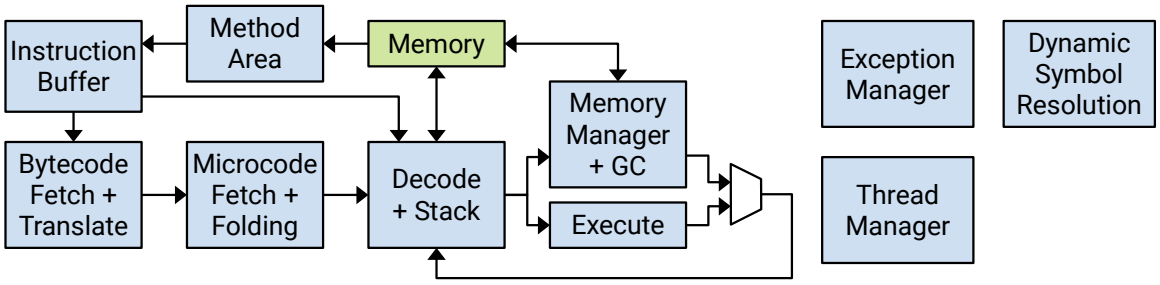


Figure 3.6.: Architecture of JAIP (adapted from [35] and [36])

As most services of the JVM have been implemented in hardware, the processor is much more complex than other architectures presented here. Its structure can be seen in fig. 3.6. The method area manager caches method code. Additionally, it stores method and class meta information in an internal memory. These memories limit the maximum number of classes and methods per program. The first stage of the 4-stage pipeline fetches bytecode into an instruction buffer, which provides inline operands to the decode stage. Fetched instructions are treated either as simple or as complex. Simple instructions are translated into microinstructions directly. Complex instructions are translated into addresses for the microcode sequence ROM. Up to two instructions are fetched and translated simultaneously. The next stage also handles both types of instructions differently. Two simple instructions can be folded into a single microinstruction. If folding is not possible, the instructions are sent to the next stage in two consecutive cycles. Complex instructions trigger fetching from microcode sequence ROM. Folding is not required for

them. The next stage decodes microinstructions and reads operands from the stack. The last stage executes the instructions and writes results back into the stack.

The stack implementation must provide multiple simultaneous accesses because of instruction folding. Therefore, the stack is composed of an on-chip memory architecture and three registers for storing the top operand stack values. Additionally, four registers store the first LVs of a method frame. The stack memory architecture is divided into two stacks to enable fast thread switching. Only one of both stacks is used actively by the pipeline. The other one is swapped to external memory and is filled with the contents of the next thread in background. The memory manager provides access to heap memory through a data cache. A hard-coded garbage collector is available. At the time of publication, this garbage collector is very limited because it does not trace which objects can be reached indirectly by object fields.

Three blocks have no connections in fig. 3.6 because their interaction with the rest of the system is more complex. The exception manager searches for exception handlers. The lookup table required for this is fully stored in on-chip memory, which limits the maximum number of exception handlers in a program. The thread manager is responsible for thread scheduling and synchronization. Additionally, it controls swapping of stacks. As explained in section 2.1.2, classes, methods, and fields are referenced by symbolic names in class files. Most other Java processors replace these references by resolved addresses either during static linking or at runtime after first resolution. JAIP keeps the original instructions instead and resolves references repeatedly at runtime. This is done by the dynamic symbol resolution unit. It contains a small cache for storing resolved addresses similar to a translation lookaside buffer (TLB) in traditional memory management units.

This processor has been compared against a JIT-based JVM on a PowerPC processor with Linux operating system using JemBench [37]. Both processors run at the same clock frequency of 100 MHz. JAIP turns out to be 30 % slower for single-threaded applications. However, the JIT compiler optimizes code while JAIP executes raw unoptimized code. Furthermore, JAIP achieves a speedup of 17 when multithreading with 16 threads is used because of much faster thread switching and synchronization.

3.1.7. Summary

The Java processors presented here can be grouped into two classes. The first class is designed for hard real-time requirements and tight resource constraints. Jamuth, JOP, and SHAP belong to this class. The second class is designed as full-featured JVM substitution. Examples are picoJava-II, aJ-200, JAIP, and the AMIDAR processor as described in section 2.2. Table 3.1 gives a detailed comparison of Java processors in terms of features, microarchitecture, size, and performance. However, exact comparison of hardware requirements and performance is rather difficult across different implementation technologies if precise numbers are published at all. The numbers in this table serve only as rough hints. Obviously, JOP and SHAP have small resource requirements. picoJava-II and JAIP have medium requirements. AMIDAR consumes by far the most resources.

Fair benchmarking is especially complicated due to different capabilities. Complex benchmarks like SPEC JVM98 [39] cannot be run on most processing systems. Apart from AMIDAR, only picoJava-II should theoretically be able to execute these benchmarks but no numbers have been published. Only the application benchmarks of JemBench [37] have been evaluated on a larger number of systems. They have been developed for small

	picoJava-II	Jamuth	JOP	SHAP	aJ-200	JAIP	AMIDAR
Pipeline Stages	6	5	4	4	?	4	No pipeline
Instruction Folding	4 instructions	✗	✗	✗	?	2 instructions	✗
Microcode	✓	✓	✓	✓	✓	✓	✓
Stack Design	64 regs as cache (incl. LV)	Cache (excl. LV)	On-chip memory (incl. LV) + 2 regs for TOS	On-chip memory (incl. LV) + 2 regs for TOS	Combined cache + 6 regs for TOS	On-chip memory (incl. LV) + 3 regs for TOS + 4 regs for LV	On-chip memory (incl. LV) + 2 regs for TOS
Other Caches	Code, heap	Code (cache and scratch)	Code (whole method)	Code (whole method)	Combined code, heap, stack	Code, meta info, heap	Code, meta info, heap
Exception Handling	S	S	S	M	?	H	H
Heap Management	S (H: handles + write barriers)	S	S	H	S	H (limited GC)	H
Scheduling	S	H	S	M	M	H	H
Synchronization	S (H: 2 lock counters)	S	S	M	M	H	H
Hard Real-Time	✗	✓	✓	✓	✗	✗	✗
Invocations	M (S: interface)	S	M	M	?	H	H
Dynamic Class Loading	✗	✓	✗	✓	✗	✓	✗
Mul/Div/ Long/FP	H/H/M/H	?/?/?/S	H/S/M/S	?/?/✗/✗	H/?/M/H	?/?/H/H	H/H/H/H
Java Support	J2SE	J2ME/CDC (almost)	Custom real-time profile	J2ME/CLDC	J2ME/CDC	?	Java SE 8 (limited API classes)
Technology	Cyclone EP1C6Q240	?	Cyclone EP1C6Q240	Spartan3 XC3S1000	0.18 μm ASIC	Kintex 7 XC7K325T-2	Artix 7 XC7A200-1
Frequency	40 MHz	33 MHz	100 MHz	50 MHz	180 MHz	100 MHz	100 MHz
Size	27543 Logic Cells, 47.6 KiB BRAM [38]	?	3300 Logic Cells, 7.6 KiB BRAM	2387 Slices, 3 DSPs, 22 KiB BRAM	?	22266 LUTs, 11366 FFs, 304 KiB BRAM	60242 LUTs, 34837 FFs, 65 DSPs, 560 KiB BRAM
JemBench							
Kfl	23813	3400	19907	11570	25466	30681	10611
UdpIp	11950	1500	8837	5764	11547	16078	5843
Lift	25444		18930	12226		30173	11628

Table 3.1.: Feature comparison of Java bytecode processors. Parity bits have been ignored for BRAM capacity. JemBench scores for the first five processors have been taken from [32]. Scores for aJ-200 have been scaled by clock frequency. (TOS: Top of Stack, LV: Local Variables, H: Hardware, M: Microcode, S: Software)

real-time systems. The *Kfl* and *Lift* benchmarks are pure control applications. *UdpIp* runs a lightweight UDP/IP stack. These benchmarks do not even closely exhaust JVM features. They create only very few objects, execute mostly static calls, do not use floating point or 64 bit numbers, and have only a single thread. Consequently, they allow only performance estimation of basic features instead of comprehensive comparison. Exact scores are given in table 3.1. They describe the numbers of benchmark iterations per second (higher is better). Scores for picoJava-II, Jamuth, JOP, and SHAP have been taken directly from [32]. The same source contains numbers for aJ-100, which runs at 100 MHz. These scores have been scaled to the clock frequency of aJ-200. Results for JAIP have been taken from [36]. Results for AMIDAR have been measured. Figure 3.7 illustrates these numbers relative to AMIDAR.

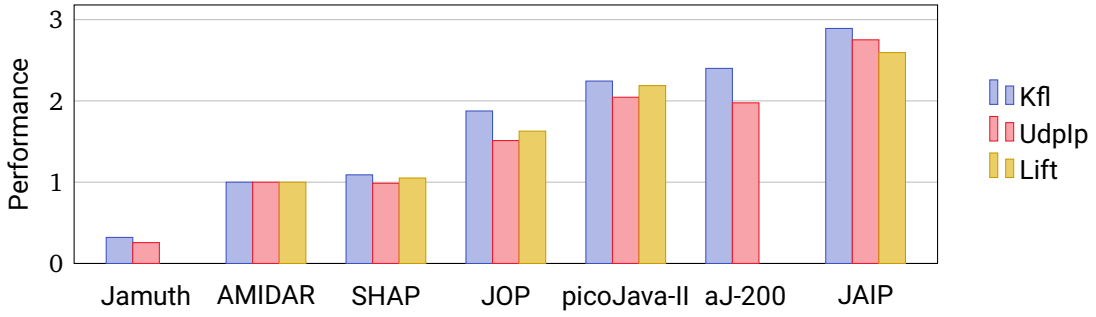
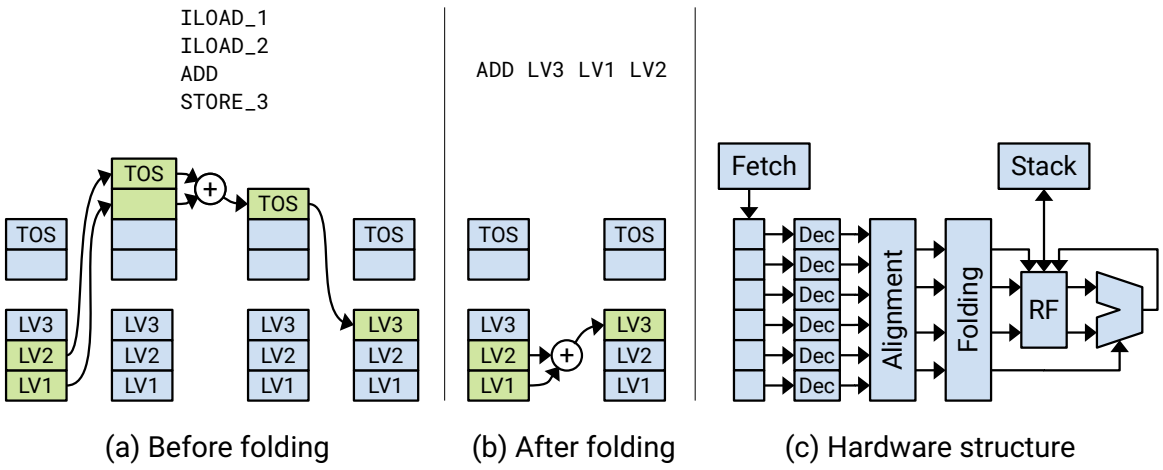


Figure 3.7.: JemBench application benchmark performance relative to AMIDAR [32, 36]

AMIDAR is the only presented processor which forgoes the RISC-like processing pipeline in favor of more freedom for runtime adaptivity. Although JemBench application benchmarks show a very limited picture, the performance numbers for AMIDAR are a bit disappointing. Merely Jamuth is slower. CGRA acceleration does not provide significant speedup for these control flow dominated applications either. One can assume that the AMIDAR microarchitecture is not ideal for stack-based ISAs. Static execution pipelines seem to be a better choice. However, the pipeline concept for Java processors has already been researched extensively. Consequently, the opposite approach is more promising as research direction: Can Java bytecode be replaced by another ISA which exploits the AMIDAR microarchitecture better? Before this question can be answered, it is worth looking at techniques to accelerate bytecode execution as well as at other computing architectures which do not use static processing pipelines.

3.2. Accelerated Execution of Java Bytecode in Hardware

A stack machine like the JVM uses separate instructions for specifying operations, operands, and result targets. Figure 3.8a shows a bytecode sequence for adding two LVs. LVs 1 and 2 are pushed onto the top of stack (TOS) by the first two instructions. The ADD instruction pops these two values from the stack, adds them, and pushes the result. Then, the result is popped from the stack and stored into LV memory. If the hardware could operate on LVs directly, data transfers and clock cycles could be saved. Two basic techniques have been proposed to take advantage of this opportunity.



(a) Before folding (b) After folding (c) Hardware structure

Figure 3.8.: Instruction folding example (adapted from [26])

3.2.1. Instruction Folding

The first technique, instruction folding, has been introduced by picoJava [26]. The basic idea is to detect a set of related instructions (folding group) in hardware and to execute them as single, combined RISC instruction. This is illustrated in fig. 3.8b. The required hardware structure is shown in fig. 3.8c. Instructions are fetched into a buffer. Each Java bytecode instruction is composed of the 1 B operation code followed by a variable number of bytes which serve as arguments. Therefore, the instruction buffer is segmented into bytes. All bytes are decoded in parallel into internal operation codes, instruction types, and instruction lengths. Then, instructions are aligned using length information. This step allows to distinguish between operation codes and argument bytes. Finally, incoming instructions can be converted into a single, internal instruction depending on instruction types. This folded instruction has the same structure as instructions of register-based RISC machines and can be executed by a similar data path. A register file (RF) as stack cache provides random access to multiple operands. Folding is permitted only if the required operands do not lie too far away from the top of stack so that they are present in the register file.

Different categories of instruction folding strategies exist [40]. Continuous folding can handle only folding groups which consist of directly adjacent instructions whereas discontinuous folding can fold groups which are interrupted by other instructions or groups. Pattern-based approaches fold a limited amount of pre-defined patterns based on instruction types whereas rule-based approaches apply rules iteratively to determine foldable sequences. The maximum number of instructions in a folding group is an important parameter of all folding techniques.

picoJava-II [27] uses a continuous, pattern-based strategy. Up to four instructions can be folded at once. Instructions are grouped into six types. Nine folding patterns are available. The POC mechanism [41] is a continuous, rule-based approach. Three types of instructions (producer, operator, consumer) exist. The rules can be represented as state diagram. Each instruction induces a transition depending on its type. However, this state diagram is not implemented sequentially but combinationally as folding cascade. Although folding groups could be arbitrarily large in theory, they have been limited to 4 instructions. This mechanism has later been extended by software-driven reordering of instructions to

improve efficiency [42]. APOC [43] is a discontinuous, pattern-based technique. It enables out-of-order issuing of folding groups but requires additional checking of dependencies between groups. Up to 6 instructions can be folded. EPOC [40] is a discontinuous, rule-based technique. If producer instructions such as local variable loads are not required by operator or consumer instructions immediately, they are stored in a stack reorder buffer. Then, operands for the execution stage are fetched from this buffer instead of the operand stack. Up to four instructions can be folded per cycle. The reorder buffer can store eight entries.

	picoJava-II	**POC**	**EPOC**
Folded Stack Ops	42 %	83 %	99 %
Speedup	1.25	1.54	1.74

Table 3.2.: Comparison of instruction folding strategies [40]

[40] provides a comparison of picoJava-II, POC, and EPOC strategies. Bytecode traces have been generated while executing benchmark programs on a JVM. These traces have been analyzed for folding opportunities according to the chosen strategy. Resulting execution times have been estimated with a simplified timing model. Table 3.2 contains the results. The best folding technique, EPOC, removes almost all instructions which only copy data from or to the stack. A speedup of 1.74 can be achieved. Only the instruction folding strategy of picoJava-II has been implemented in a real processor. All the others have been evaluated by analysis of bytecode traces. Hence, consequences of increased hardware complexity are not taken into account. Instruction folding has turned out to be on the critical path in picoJava-II [44]. Implementation of a continuous, pattern-based mechanism in another Java processor [7] has shown that instruction folding decreases clock frequency and increases hardware requirements significantly, which has even resulted in reduced performance. This would become even worse if more powerful folding strategies were used.

3.2.2. Synthilation

Synthilation (a neologism derived from “synthesis” and “compilation”) [45] is another technique for accelerated execution of Java bytecode by eliminating unnecessary stack accesses. In contrast to instruction folding, it avoids complex hardware modifications. A runtime profiler finds code sections which consume a high amount of execution time. A separate software thread compiles these sections into optimized microcode sequences. The original code is patched with custom instructions which trigger these optimized sequences. The only modified hardware components are the additional profiler and writable microcode memory of increased size. A simulated speedup of 1.56 has been achieved. This technique comes with further advantages which allow even higher speedups. More sophisticated optimizations can be used compared to instruction folding in hardware. Furthermore, additional computing hardware can be exploited. However, synthilation can be applied only to parts of a program because microcode memory is very limited.

In theory, one could go one step further by applying synthilation to the whole program before execution, which would effectively replace Java bytecode by a customized ISA. Nevertheless, exposing all microarchitecture details would increase code size drastically

and forbid runtime reconfiguration. Therefore, the better choice would be an ISA which focuses particularly on the removal of inefficiencies caused by the operand stack model. Such an architecture could potentially achieve the same or even higher speedups than instruction folding or synthilation while reducing hardware effort to a minimum. Therefore, it is advisable to study different ISAs. Since AMIDAR synchronizes FUs by data transfers internally, one should have a special look at architectures which describe data flow explicitly. This might yield ideas for such a customized ISA.

3.3. Data Flow Architectures

Theoretical foundations for data flow architectures have been laid by Karp and Miller [46] who have expressed algorithms by directed graphs. In the following years, the concept has been extended to describe whole programs by data flow graphs [47, 48, 49]. Nodes of such graphs represent operations. Edges define how "tokens" can "flow" from one operation to another. The order in which tokens flow along an edge is maintained. Thus, edges behave like unbounded queues. Each token carries a data value. Any time after tokens are present on all incoming edges of a node, the operation "fires" and places tokens on its outgoing edges. There is no total order in which operations fire. In order to avoid confusion, it should be noted that tokens in the context of data flow graphs and architectures are completely different from tokens in the context of AMIDAR.

```
sum = 0;
for (i = 0; i < 5; i++)
   sum += i;
```

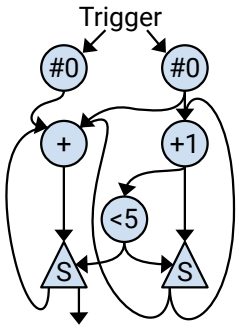


Figure 3.9.: Data flow graph for a loop (adapted from [50])

Figure 3.9 depicts a data flow graph for a loop which computes the sum of the integers from 0 to 4. Initial trigger tokens are required to start execution. The right half of the graph increments the counting variable, the left half sums up these values. Decisions are realized using *switch* operations. They forward incoming data values to one of the outgoing edges depending on received control values. If the end of the loop is reached, the final sum leaves the loop on the right outgoing edge of the left switch operation. The right switch operation does not produce output in this case.

Operations are purely functional and execute without side effects in the pure data flow model. There is no concept of a memory which stores the state of a program centrally. All state information is stored in tokens. Theoretically, elements of data structures and arrays can be represented as individual data values. Alternatively, all required data structures can be contained fully in tokens. This implies copying unchanged elements from one token to another because tokens are immutable. This is often called the *single assignment* property.

Data flow computing architectures use such data flow graphs as ISA. In contrast to control flow based von Neumann architectures, they have neither a fixed instruction

execution order nor a global memory. While data dependencies are defined implicitly by the order of memory accesses in the von Neumann model, they are defined explicitly by instructions in the data flow model. This promises two main advantages:

- Because no sequence is imposed by the programmer, operations can be executed in parallel to a large extent. Thread level parallelism (TLP), data level parallelism (DLP), and instruction level parallelism (ILP) can exploited easily.
- As execution is asynchronous by nature, dynamic latencies caused by operation execution, data transmission, or external data access can be handled without additional effort. Furthermore, latencies can be masked by other operations.

This section gives a short overview of different data flow architectures. Descriptions have been simplified and technical terms have been harmonized to facilitate understanding and comparison. Memories are marked green in the following block diagrams.

3.3.1. MIT Static Data Flow Machine

One of the first data flow machines has been developed by Dennis at MIT [51, 52]. Figure 3.10 shows its architecture. A token contains a data value together with a target. This target is composed of the address of the instruction which uses the value as operand and a port number which allows to distinguish between multiple operands of one instruction. Cell blocks are the most important elements. Each of them contains an activity store. This memory stores the instructions together with their operands. One storage location and presence bit is provided for each operand of an instruction. When a token is received by the update unit, the value is stored at the specified position in the activity store. The unit also checks if all operands of the instruction are present. If this is the case, the instruction address is added to the instruction queue. The fetch unit removes addresses from this queue, loads the corresponding instructions with their operands from the activity store, and sends them to FUs for execution. Apart from the operation code, an instruction contains one or two targets. An FU generates a new token containing the result of the computation for each target. These tokens are sent to the cell blocks again where they trigger the next instructions. Replication of data is realized by specifying two targets in an instruction. If a higher fan-out is required, a tree of duplication instructions must be created.

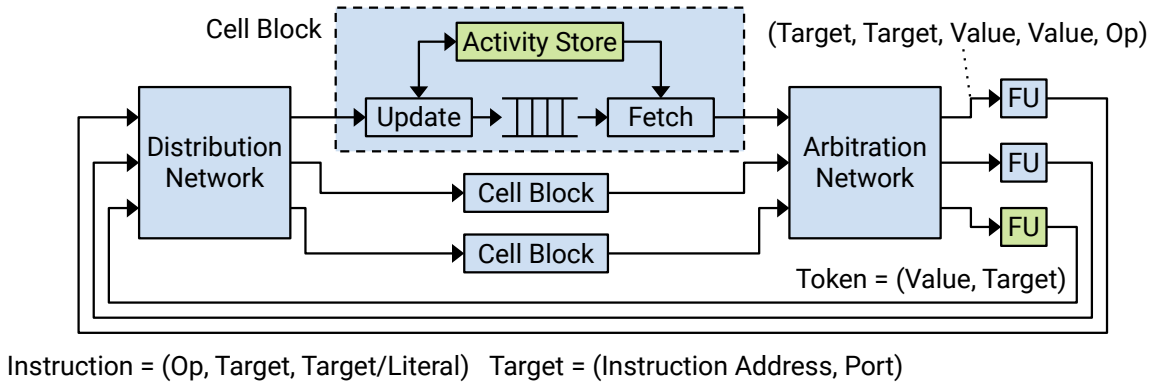


Figure 3.10.: Architecture of MIT static data flow machine (adapted from [52])

Because only a single location exists for each operand in the activity store, a new operand must not be produced before the previous one has been consumed. This is ensured by special acknowledge signals. Hence, the parallelism which can be exploited in loops is limited and recursion is not possible. Such an implementation is called a *static* data flow machine.

Activity stores must provide access to all instructions of a program together with their operands. Larger programs do not fit in these memories. Consequently, a memory hierarchy with caching mechanisms must be applied. However, caching is more difficult in data flow architectures because temporal and spacial locality is lower due to non-sequential execution.

Storage for data structures is provided by a special FU. Only the pointer to a structure is put into a token instead of the elements. However, such a pointer cannot point directly to space in memory. Multiple versions of the same structure might be required concurrently because of non-sequential execution of instructions. This problem is solved by implementing structures as trees [53]. A pointer to a structure points to the root node. The leaf nodes are the elements of the structure. If data is written to an element, a new tree is created which shares as many nodes as possible with the previous tree. A new leaf node is created only for the element which has been written. All other leaf nodes are identical. The pointer to the new tree is returned. Tokens which hold the previous pointer can still be use to read the previous value of the element. Single assignment semantics can be provided this way. Reference counting is utilized for cleaning up obsolete trees or parts of it. The major disadvantage of this approach is the overhead in terms of memory size and access latency.

3.3.2. Manchester Machine

To exploit the full amount of parallelism, *dynamic* data flow machines allow multiple tokens to be present on each edge of the data flow graph at the same time. The architecture of the Manchester machine [54, 55] is shown in fig. 3.11 as an early example for dynamic data flow machines. A tag is added to tokens in order to distinguish between multiple tokens on the same edge. This avoids enforcing FIFO semantics for each edge. A tag encodes the procedure number, which distinguishes between multiple invocations of the same procedure, and the iteration count, which distinguishes between iterations of the same loop. In addition, target information contained in instructions and tokens specifies a match type. It defines how many operands the target instruction requires. Tokens are buffered in a queue before entering one of the matching units. Their task is to match the operands of each instruction. If a token is targeted to an operator with one operand, the token passes the matching unit directly. Otherwise, the second operand is searched in the hash table. If it is available, both operands are sent to the instruction fetch unit. If it is not available, the current operand is stored into the hash table where it waits for the second operand. The instruction fetch unit loads the corresponding target instruction. Afterwards, the operation is executed by an FU. A new token containing the result is generated for each target specified in the instruction.

A special FU, the procedure processing unit (PPU) manages procedure calls. The array processing unit (APU) provides storage for data structures. Again, data structures are referenced by pointers. Structures are stored as contiguous space in memory initially, which allows for fast access and efficient memory usage. Changes to elements are not

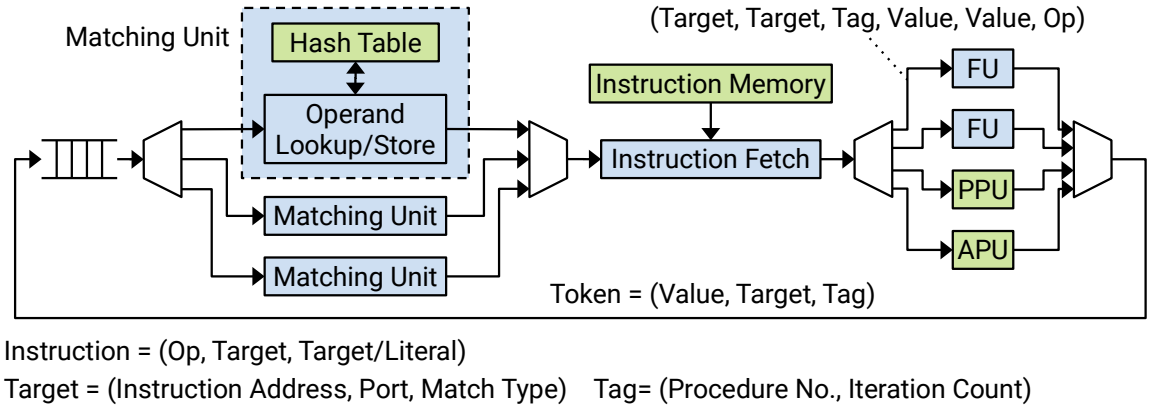


Figure 3.11.: Architecture of Manchester machine (adapted from [55])

stored in this space. To fulfill the single assignment property, they are stored in form of linked lists which finally point to the original structure. A structure is referenced by a pointer to the beginning of such a list. The list is traversed on read accesses and the latest (first in the list) change to the requested element is returned. Reference counting is utilized to detect whether certain states of the structure are not referenced any more and can be merged into the original memory space. This technique still has significant overhead in terms of read latency if many states of a structure are referenced concurrently.

Evaluation shows that this architecture is capable of using many FUs efficiently if the program provides enough parallelism. The main problem is the matching mechanism, which requires complex hardware because of the associative search of operands. All working data must be stored here except for data managed by the APU. A memory hierarchy for hash tables slows them down even more. The problem is aggravated by too much parallelism in some programs. For example, if a loop has many iterations which are independent from each other, this loop will be unrolled fully. This results in a large amount of tokens which must be stored in hash tables. Another problem is the overhead caused by data replication, which accounts for 25 % of all executed instructions.

3.3.3. Monsoon

Another dynamic machine which manages operand matching without expensive associative search is Monsoon [56]. It is based on the MIT tagged-token architecture [57]. Its structure is illustrated in fig. 3.12. While MIT static data flow machine and Manchester machine are built in ring structure with some replicated components, this architecture replicates whole PEs each with an autonomous processing pipeline. Tokens are used for communicating among PEs. An *explicit token store* is introduced to facilitate operand matching. A frame, which serves as context for tokens, is allocated dynamically for each invocation of a code block (procedure or loop iteration). Each frame is assigned to one PE only. Therefore, the tag of a token consists of PE number and frame pointer. Matching can happen statically inside a frame, which allows to abandon associative memory. The compiler can allocate a synchronization slot in the frame for each instruction which requires two operands. Slots can also be re-used if instructions cannot be executed concurrently because one depends on the other. Von Neumann architectures typically build a stack of frames (one for each procedure) where only the top frame is active. Monsoon builds a tree of frames instead

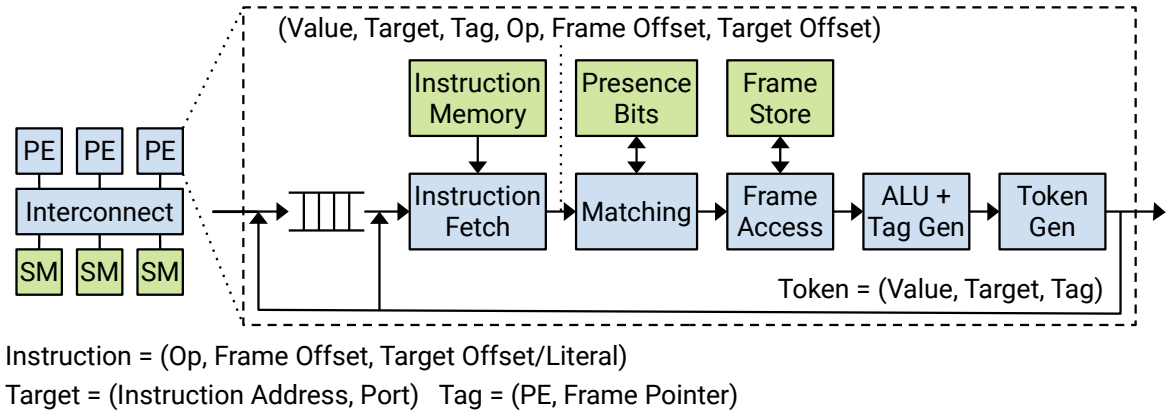


Figure 3.12.: Architecture of Monsoon (adapted from [56])

where multiple frames can be active concurrently.

The processing pipeline of a PE is shown in fig. 3.12. Incoming tokens are buffered in a queue. Afterwards, the instruction belonging to the instruction address of the token is fetched. The instruction contains a frame offset. If it is added to the frame pointer of the token, it points to the synchronization slot of the instruction. A synchronization slot consists of presence bits and storage for one operand value. If the instruction has only one operand, matching and frame access is skipped. Otherwise, presence bits are checked. If the second operand is available, it is read from the frame store and passed to the ALU together with the value of the token. Otherwise, this value is written to the frame store. The ALU executes the computational part of instructions. Tag and target are updated in parallel. Finally, one or two tokens are formed and are sent to the PEs which are specified in their tags. A direct feedback path exists for tokens which target the local PE. One target can be the instruction which follows the current one in the instruction memory immediately. The second target instruction is determined by adding the target offset contained in the current instruction to the current instruction address.

The problem of congestion caused by unrestricted loop unrolling is avoided by inserting artificial data dependencies. They limit the number of iterations which are processed in parallel. This technique is called *loop throttling*.

Apart from PEs, the processor contains structure memories (SMs) for storing *I-structures*. I-structures are arrays in which each element can be written only once. If an element is read before it is written, the read access is deferred until the write access has finished. Data dependencies are enforced on array elements this way. Access to I-structures is faster than to trees or to lists of changes.

Several flaws have been identified. Allocating and deleting frames causes significant overhead (around 50 % of executed instructions). Replicating data and detecting if a frame can be deleted are other sources of overhead (about 39 % of clock cycles). Furthermore, the ALU is unused in 29 % of the clock cycles because it is waiting for a second input operand.

3.3.4. Epsilon-2

Replicating data values efficiently is an important problem of pure data flow architectures. Furthermore, they suffer from poor pipeline utilization if a program provides few

parallelism. If one instruction sends its result to a second instruction, the second instruction cannot enter the processing pipeline before the first instruction has left the pipeline. Forward-looking instruction issuing is not possible because a fixed instruction sequence is missing. These two problems are addressed by the hybrid architecture Epsilon-2 [58]. *Grains* of instructions are allowed to be executed sequentially in addition to pure data flow processing. Only the first instruction of a grain is scheduled by data dependencies. A register file can be used to pass data between instructions within a grain.

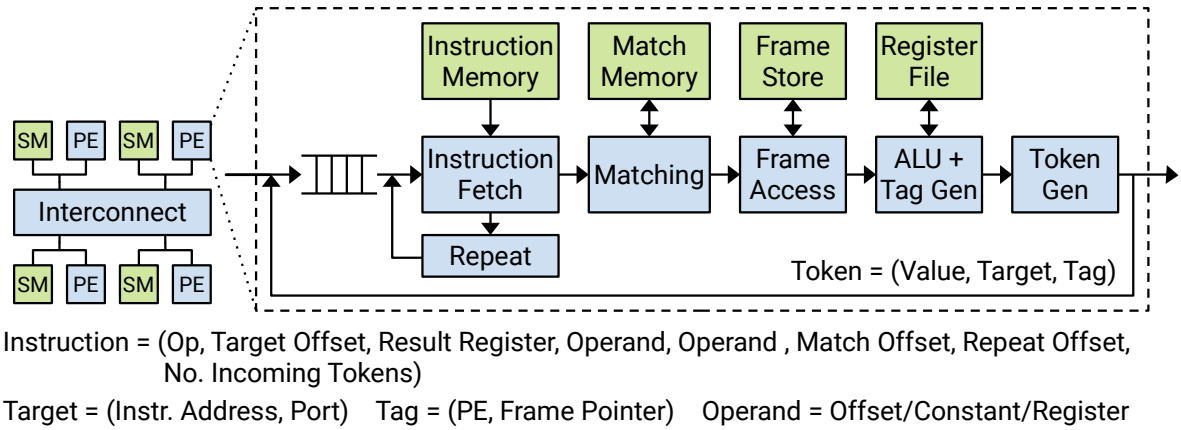


Figure 3.13.: Architecture of Epsilon-2 (adapted from [58])

The architecture depicted in fig. 3.13 has many similarities to Monsoon. A major difference in data flow processing is the ability to wait for more than two tokens per instruction. This is achieved by replacing presence bits by an operand count in the match memory. If a token is received, this count is incremented for the target instruction. The instruction is executed when the count equals the number of expected incoming tokens. A repeat unit is added to allow sequential execution of instructions. If an instruction contains a repeat offset, a new token is generated at the input of the pipeline immediately. This token contains the same contents as the previous token except for the instruction address, which is incremented by the repeat offset. This simplifies using one token as operand for multiple instructions. Furthermore, a register file is added which can be read or written by the ALU. Every instruction specifies a register for its result. Operands can be taken from constants, the frame store, or any of the registers.

Every PE is paired with one structure memory (SM) which can be accessed faster than the other SMs. They support I-structures, traditional arrays, and lists. However, it is not explained how arrays and lists can be used safely with data flow scheduling.

3.3.5. TRIPS

The architectures presented so far have been developed by improving the data flow concept incrementally. They all have an important drawback: They cannot be programmed by traditional procedural programming languages because they do not support the von Neumann memory model. Special data flow languages which are based on the functional programming paradigm are used instead. In contrast, TRIPS [59, 60, 61] is a hybrid architecture developed as an alternative for very large instruction word (VLIW) processors or out-of-order superscalar processors. Hence, it provides full support for the procedural programming model.

A program is grouped into atomic *hyperblocks*. A hyperblock can contain up to 128 instructions but only a single jump, which is executed after the block. The compiler forms large hyperblocks with the help of predication and loop unrolling. Hyperblocks are scheduled by sequential control flow and they transfer data using register files. The block header specifies these live-in/out registers. In contrast, instructions specify data flow within hyperblocks explicitly, similar to pure data flow architectures. Instructions are executed on an array of PEs. The spatial positions of instructions are defined by the compiler in terms of rows, columns, and frames. Distances of data transfers are tried to be minimized. Data dependencies determine when an instruction is executed. The static data flow execution model without tags is applied.

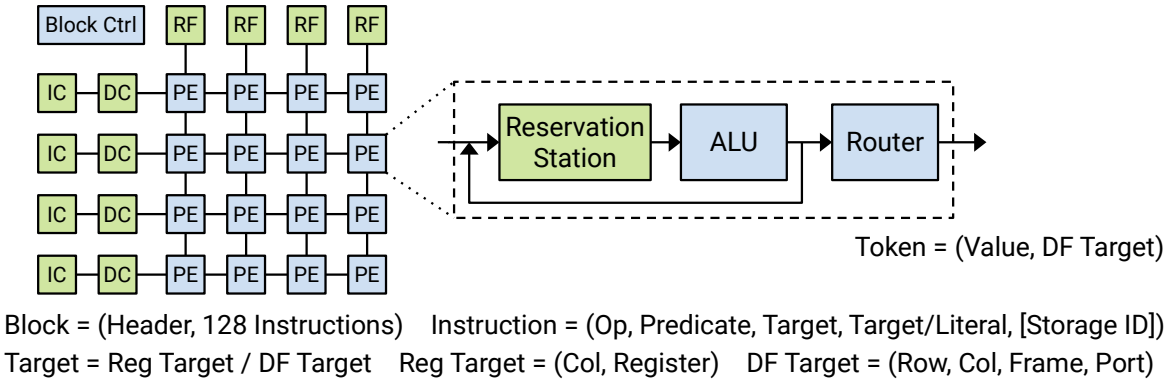


Figure 3.14.: Architecture of TRIPS (adapted from [60])

The architecture is shown in fig. 3.14. It contains a 4x4 array of PEs. Each of them consists of a reservation station, an ALU, and a router. A reservation station can hold up to 64 instructions (8 per hyperblock). Two operand slots are available for each instruction. As soon as all operands of an instruction have been received, it is executed by the ALU. The router sends the result to one or two targets specified in the instruction. A direct path to the input of the reservation station is utilized if the result targets the local PE. The instructions in all PEs which have the same address within each reservation station form a frame. Consequently, a data flow target is defined by row, column, frame, and port. Only complete blocks are loaded from instruction caches (ICs) into reservation stations. Eight blocks can be loaded simultaneously. Each column has one register file for communication between hyperblocks. They support forwarding of interblock values, which allows multiple blocks to be executed concurrently. There is one L1 data cache (DC) per row. They provide a generic global memory. Ordering of memory accesses is enforced by IDs in load and store instructions within hyperblocks. Global control for loading hyperblocks is located in one corner. Different strategies for block loading exist to focus on ILP, DLP, or TLP.

The prototype shows a speedup of 3 compared to a Core 2 processor with hand-optimized kernels but achieves only 60 % of the performance for compiled, complex benchmarks. Apart from weaknesses in the compilation process, several architectural problems have been identified. One of them is overhead of binary encoding, especially for small blocks and complex call graphs. Another problem is inefficient data replication as in many other data flow architectures. Cache and register bandwidth is insufficient as well.

3.3.6. WaveScalar

WaveScalar [50] is another example for hybrid architectures which fully support von Neumann memory semantics and procedural programming languages. It is designed as an alternative for out-of-order processors with focus on improved scalability. In contrast to TRIPS, it does not make use of sequential control flow scheduling of blocks. The compiler partitions programs into *waves*. Every instruction in a wave must not be executed more than once. Consequently, a wave does not contain loops. Additionally, a wave may have only a single starting point. The tokens of this architecture are tagged with wave numbers, which are incremented between waves by special instructions. Hence, WaveScalar is a dynamic data flow architecture. Instructions define the targets of their results explicitly. However, no details about instructions are provided, e.g. the number of targets per instruction. Loop throttling is applied like with Monsoon to avoid congestion by too much parallelism (see section 3.3.3). A sophisticated technique has been invented to provide von Neumann memory semantics despite of pure data flow scheduling. The compiler assigns sequence numbers to memory accesses inside a wave. Memory accesses also contain the sequence numbers of preceding and succeeding accesses. This allows to handle branches in waves, which prohibit a total ordering of accesses. Dynamic reordering of independent read accesses is supported as well.

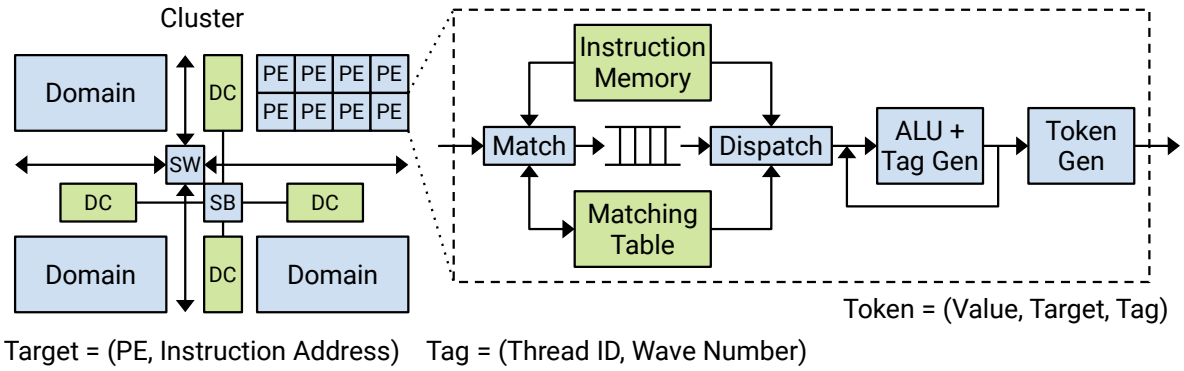


Figure 3.15.: Architecture of WaveScalar (adapted from [50])

The processor is structured hierarchically as shown in fig. 3.15. The largest unit is a cluster, which consists of four domains with eight PEs each. Additionally, a cluster contains a switch (SW), which provides connectivity between domains and clusters, and a store buffer (SB), which orders accesses to L1 data caches (DC). When a token enters a PE, the corresponding target instruction is fetched from the instruction memory. 64 instructions can be cached directly in a PE. Operands are matched similar to the technique of the Manchester machine (see section 3.3.2). In contrast to the Manchester machine, a matching table has to match operands only for instructions which have been mapped to its PE. Matching tables in PEs are caches for a larger table in external memory. The address of an instruction which is ready to execute is placed in the scheduling queue. The dispatcher removes it from the queue. Then, it loads the instruction from the memory and operands from the matching table. The ALU executes the instruction and generates the tag for the result. Finally, tokens are generated and sent to their target PEs. If a dispatched instruction targets the local PE, this target instruction is dispatched speculatively afterwards. In this case, the computation result is fed back to the ALU inputs. If the second operand has not been available yet, computation is canceled. This mechanism increases pipeline utilization

for programs with low parallelism. The compiler groups instructions which depend on each other and therefore cannot be executed simultaneously into segments. When an instruction is required for matching, the whole segment is loaded into a PE. Afterwards, the segment is fixed to this PE.

Single thread performance is evaluated by simulations of this architecture and of an Alpha EV7 processor. However, performance is compared in terms of instructions per cycle (IPC), which is problematic when using different ISAs. WaveScalar is faster in some benchmarks and slower in others. Especially many function calls reduce performance significantly. When comparing estimated chip area of both systems, WaveScalar achieves better IPC/area. It must be noted that most chip area of a cluster is used for the matching stage (43 %). Furthermore, the system scales well with increasing number of threads. It outperforms other experimental chip multiprocessors clearly when executing many threads.

3.3.7. Out-of-Order von Neumann Machines

Data flow scheduling is also used in superscalar out-of-order processors [62, 63, 64] although they appear as traditional von Neumann machines from the programmer's perspective. Register renaming creates an instruction stream internally which satisfies the single assignment rule of data flow architectures. Then, execution on ALUs can be triggered by data dependencies within a limited instruction window. Nevertheless, such processors require complex mechanisms for reconstructing data dependencies and for dynamic scheduling. A large operand broadcast network is usually used, which limits scalability. Furthermore, accurate branch prediction and speculative execution is required to fill the instruction window. Misprediction entails expensive unrolling of executed instructions.

3.3.8. Summary

The advantages which have been expected at the beginning of this section have been proven to be true. Data flow architectures can exploit different kinds of parallelism without special effort. However, their execution model comes with severe disadvantages which have already been pointed out by previous publications [65, 66]:

- Scheduling instructions by data dependencies is more complex than just executing them sequentially. This makes data flow architectures less efficient if programs do not exhibit sufficient parallelism.
- Data replication is another source of inefficiency. Register-based architectures can simply write data words into registers and read them multiple times subsequently. More complicated solutions are required without a fixed sequence of instructions.
- The absence of the von Neumann memory model makes handling of data structures difficult. As common programming languages rely on this memory model, programs must be re-implemented for many data flow machines. Some hybrid architectures solve this problem but pay for it with additional hardware complexity.
- The memory bottleneck still persists. Reduced locality makes caching mechanisms even less efficient.

Despite several decades of research, data flow architectures have not succeeded commercially. While pure data flow machines have been developed in the 70s and 80s, more hybrid approaches have been presented in the 90s and 2000s. Some of their ideas are applied in microarchitecture of modern out-of-order processors successfully.

As a hardware implementation of the JVM, AMIDAR is a control flow based von Neumann architecture. However, its microarchitecture consisting of independent FUs requires matching of operands with FU operations similar to dynamic data flow machines. Therefore, it seems natural to incorporate some data flow concepts into a new ISA for AMIDAR as well. On the other side, turning AMIDAR into a real data flow architecture does not seem ideal because of the aforementioned disadvantages.

3.4. Transport Triggered Architectures

Transport triggered architectures (TTAs) have an internal structure similar to AMIDAR. They have been developed as an alternative to VLIW processors to exploit ILP while reducing hardware complexity. VLIW processors typically require register files with high port count and a complex bypass network for operands. TTAs tackle these problems by exposing data path connections to the programmer [67]. Figure 3.16 illustrates such an architecture. FUs are connected to the transport network via port registers for their operands and results. In all other aspects, they are independent from each other and from the transport network. Instructions only transfer data between ports. Operations are triggered as side effects of writing to special trigger ports. Multiple transport buses are available to allow parallel transfers, the so-called *moves*. Therefore, each instruction for an architecture with three transport buses consists of three moves. Each move specifies a source register or a literal and a destination register.

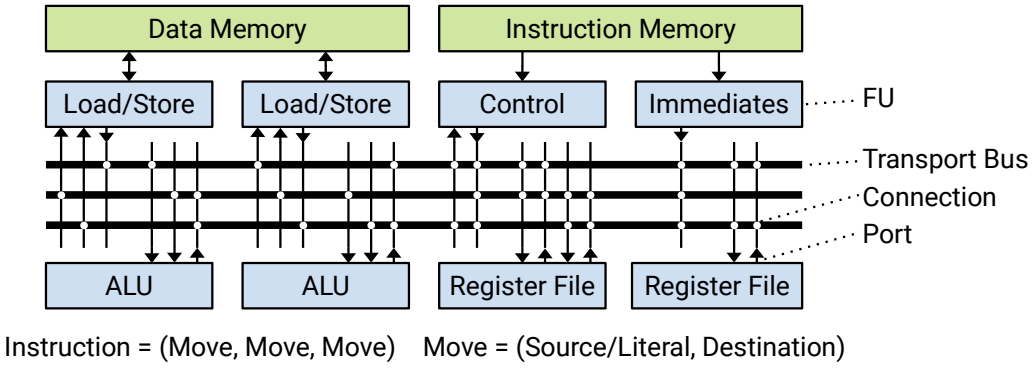


Figure 3.16.: Exemplary transport triggered architecture (adapted from [68])

This gives more freedom to the compiler. Operand and result transfers can be scheduled independently of operation execution. Furthermore, port registers can be used as temporary storage. Data can be transferred directly between them without accessing a register file. This technique is called *software bypassing*. Because these features reduce register file pressure, register files can be smaller and can have less ports. In addition, connectivity does not need to be designed for worst case usage with maximum parallelism. This improves transport network usage and allows for precise connectivity optimization. On the other side, the compiler must be more complex to exploit these opportunities. In addition, instruction size is increased significantly.

An example for TTAs is the MOVE architecture, which has been used for control processors quite early [69]. Later, it has been extended and researched extensively by Corporaal et al. [67]. Subsequently, the TTA-based codesign environment (TCE) has been introduced as toolkikt for generating and programming TTA-based application specific processors [70]. This publication compares a triple-issue TTA soft-core to a MicroBlaze RISC core and to a NIOS II RISC core. Average numbers are summarized in table 3.3. Although speedups are significant, the increase of hardware resources is even higher. Code size grows drastically. However, this TTA design has not been optimized. Additionally, comparing a triple-issue core to single-issue cores is not ideal.

TTA variant		Baseline	Speedup	Resources	Code Size
[70]	triple-issue	5-stage MicroBlaze	2.46	2.28	3.88
		Nios II/f	1.49	1.90	4.80
[68]	single-issue	5-stage MicroBlaze	1.74	1.04	1.11
	bus merged triple-issue	triple-issue VLIW imitation	1.59	0.87	0.97

Table 3.3.: Comparison of TTA soft cores generated by TCE with RISC and VLIW cores.

Several techniques for code compression have been proposed [71, 72, 73]. Furthermore, an instruction encoding has been presented which provides explicit operation codes and higher flexibility for immediate values in order to reduce code size and power consumption [74]. Additionally, it has been demonstrated that code size can be reduced without changes to the ISA by improving compilation and by optimizing the transport network [75, 68]. A single-issue TTA variant has been compared to a 5-stage MicroBlaze core and a triple-issue TTA variant with optimized transport network has been compared to a triple-issue VLIW core with partitioned register files. Again, table 3.3 summarizes average results. Resource consumption and code size increase decently. They are even reduced slightly in comparison to VLIW. However, the VLIW core is only imitated by a TTA with VLIW-equivalent transport network and instruction format. In addition, the authors note that their LLVM-based compiler performs more aggressive program optimizations than the GCC-based compiler of MicroBlaze, which influences code size considerably.

Although this architecture seems exotic, commercial TTA-based microcontrollers exist [76]. TTA and AMIDAR share the concept of independent FUs and data interconnect, which makes block diagrams look similar. Nevertheless, there are important differences. AMIDAR does not expose microarchitecture details to the programmer. Furthermore, timings of FUs and interconnect are not fixed at compile time. Hence, operations and data transfers are not scheduled statically. These properties increase design flexibility even more, tolerate dynamic latencies better, and make runtime reconfiguration feasible. TTA gains many of its advantages compared to VLIW by eliminating unnecessary register file accesses. This idea is essential for the design of a new ISA for AMIDAR. This work will show that this goal can be achieved without needing to program microarchitecture details.

Part II.

Concept

4. Problem Analysis

AMIDAR shows performance below average in comparison to other Java processors in section 3.1. As AMIDAR is the only processor which does not use a static processing pipeline, one can assume that performance problems are caused by a mismatch between ISA and microarchitecture concept. This chapter analyzes the causes of suboptimal performance carefully and defines resulting research objectives.

4.1. Weaknesses of the Current Architecture

It is worth looking at an exemplary sequence of instructions to identify performance problems. Figure 4.1 illustrates how one iteration of the loop of the running example in fig. 2.3 is executed. As semantics of these instructions have been explained in section 2.1.2, discussion will focus on aspects of execution here. Each instruction is converted into a set of tokens, which are executed on FUs. Tokens are represented by contained FU operations. Instruction and corresponding FU operations are drawn in the same color. Arrows show data transmissions between FU operations. Timing is simplified heavily because latencies for operation execution and for data transmission are ignored. In addition, an ideal instruction decoder is assumed. Some operations can have parameters which are sent to FUs along with operation codes. Such parameters are written in brackets. This is the case for loading and storing LVs in slots 0 to 3. All other constant parameters are created by `imm` operations on the Token Machine. They are transferred to target FUs over the data interconnect.

Despite the simplifications, it becomes obvious that the Frame Stack limits execution speed. By far the most operations are executed here. This is not surprising because the operand stack has been identified as bottleneck in other Java processors as well (see sections 3.1 and 3.2). The distribution of FU operations and data transmissions in fig. 4.2 confirms this assumption. These numbers have been measured on an FPGA implementation of the bytecode processor. Operations and transmissions have been summarized over all benchmarks. Details of the evaluation process will be given in chapter 12. 59 % of the operations are executed by the Frame Stack and 43 % of the data transmissions originate from the Frame Stack.

However, this does not explain yet why AMIDAR underperforms in comparison to architectures with monolithic execution pipelines. When looking at fig. 4.1 again, one can see that many instructions pop operands from the operand stack, execute some computation on another FU, and push the result back onto the operand stack. This causes many unnecessary stack operations and data transfers because it would be possible to transfer data directly between computational FUs. Unfortunately, each data transfer between FUs costs an additional clock cycle compared to a pipelined architecture. This is a drawback of partitioning a whole processor strictly into FUs. On the other hand, partitioning allows to process more operations concurrently. However, overlapping of

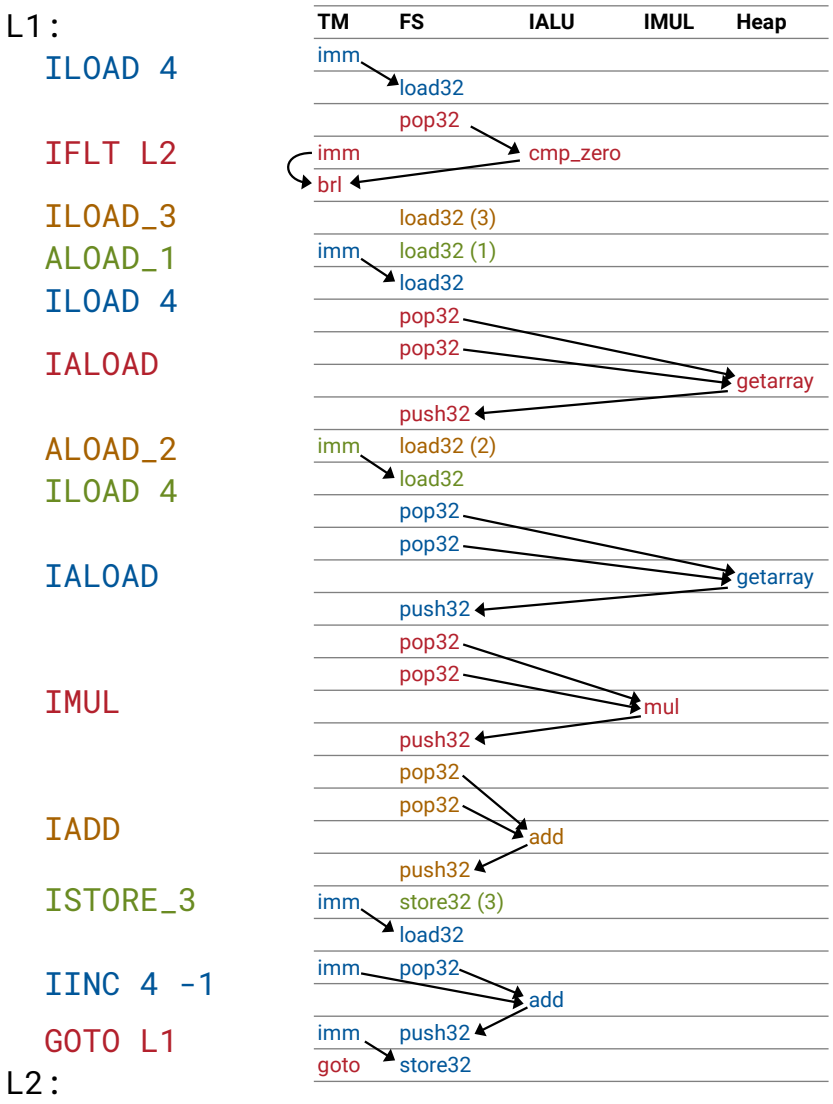


Figure 4.1.: Execution of the loop body of the running example on the bytecode-based processor

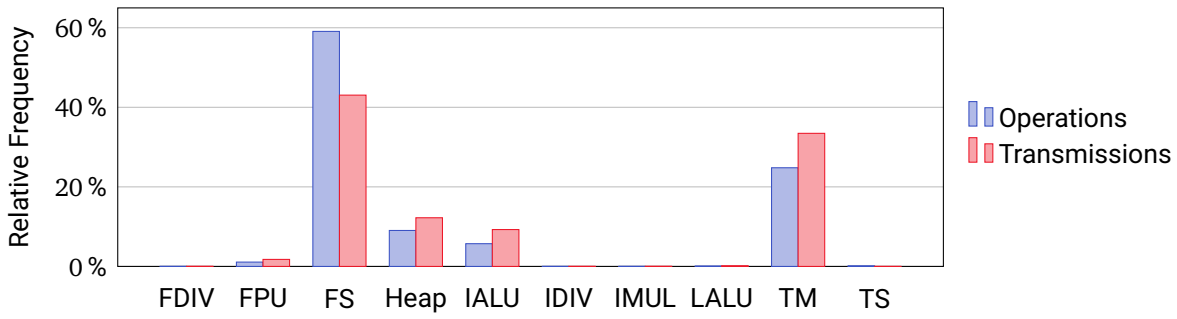


Figure 4.2.: Distribution of FU operations and transmissions for the bytecode-based processor

instruction execution is hampered by the fact that many instructions both start and end with operand stack operations. Static pipelines tackle this problem with forwarding techniques, which cannot be applied directly with AMIDAR. Consequently, the concurrency provided by independent FUs cannot be exploited. Even simple overlapping of instruction execution can hardly be applied. Constant generation on the Token Machine is the only opportunity for concurrent processing in the example.

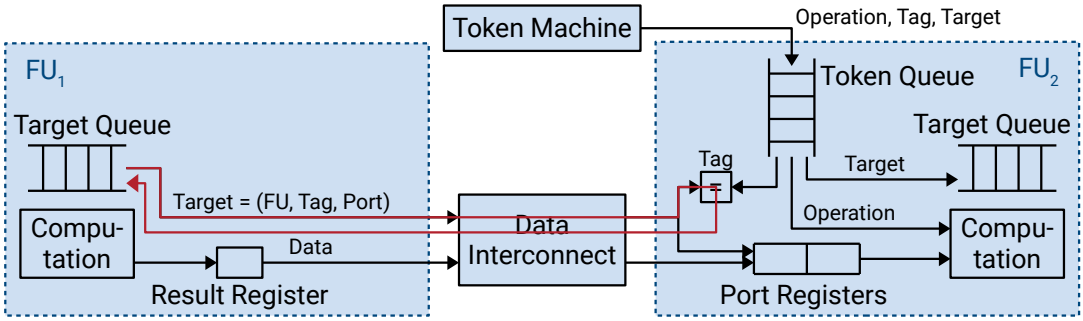


Figure 4.3.: Critical path for data transmissions in the bytecode-based processor (red)

The tag-based mechanism for matching data with FU operations is another source of inefficiency. The critical path for data transmissions is depicted in fig. 4.3. It starts at the tag output of the target queue. It runs through the data interconnect, the tag comparator, and the data interconnect again. Finally, it reaches the target queue in form of the acknowledgment signal which removes an element from the queue. A register is inserted after the comparator to improve clock frequency. Consequently, each FU can send data every second clock cycle only. This is a severe limitation in case of the Frame Stack because it often sends the two top of stack values to another FU.
Furthermore, this technique allows operands to be received only for the element at the front of the token queue. If more concurrency was provided by token sets, this could become a bottleneck and could result in many rejected transmission attempts. Consequently, this mechanism exhibits suboptimal data throughput.
Additionally, fair arbitration is required among incoming requests because only one of them has a matching tag and can be accepted. This introduces additional hardware complexity and logic delay. Currently, this is avoided by direct connections between FUs. However, if transmissions between arbitrary FUs are required for the new ISA, direct connections will probably not be feasible. To sum up, the following weaknesses have been identified:

1. Many operand stack operations and related data transfers are unnecessary.
2. Opportunities for concurrent or overlapping computations are not exploited.
3. FUs are utilized unevenly.
4. Throughput of the operand matching mechanism is too low.

4.2. Research Objectives

Weakness 4 found in the previous section is related to microarchitecture only. Weaknesses 1–3 are closely related to Java bytecode and are magnified by the combination with the AMIDAR principle. One can tackle these problems by microarchitecture improvements. Study of other Java processors in section 3.1 has shown that carefully designed pipelines are more efficient than separate FUs in terms of performance per chip area for processing bytecode instructions. However, switching to a pipeline architecture would confine adaptivity opportunities. In addition, pipeline architectures for Java bytecode processors have already been researched extensively. The potential for further performance improvements seems limited here. The operand stack becomes a bottleneck quickly which can be circumvented only by complex hardware additions like instruction folding (see section 3.2.1). Synthilation avoids these hardware additions but can be applied only to parts of a program (see section 3.2.2).

Both approaches have one thing in common: They convert Java bytecode into another ISA either in hardware or in software at runtime. This gives the idea to develop a new ISA and to move this conversion process to compile time. Consequently, the central question which will be answered in this thesis is: Can Java bytecode be replaced by another ISA which exploits the AMIDAR microarchitecture better? Nevertheless, an ISA can never be considered isolated. It is the core component of programmable computing systems. Developing a new ISA implies substantial redesign of microarchitecture and code generation. A good balance between software and hardware components must be found. Based on the strengths of the generic AMIDAR concept and the weaknesses of the specific Java bytecode implementation, the following objectives [8] should be achieved by the new ISA in conjunction with microarchitecture and software tool chain improvements:

1. The amount of FU operations which are used only to pass data between instructions should be reduced. Likewise, unnecessary data transfers between FUs should be eliminated. This can be achieved by transmitting data directly between FUs without intermediate storage.

2. More concurrent processing of instructions and FU operations should be allowed.

3. The ISA should operate on a high abstraction level similar to Java bytecode in order to assist synthesis of program parts into CGRA contexts.

4. Opportunities for runtime reconfiguration should be retained. Therefore, the assignment of operations to FUs should not be fixed at compile time. Furthermore, no assumptions about FU and interconnect timing behavior should be required, neither during code generation nor during token generation.

5. All JVM features should be supported. This includes arbitrarily complex control flow, exception handling, multithreading, and garbage collection.

6. Instruction encoding should be compact in order to avoid a bottleneck between code memory and instruction decoder. Hence, microarchitecture details cannot be exposed fully. This objective is closely related to objectives 3 and 4. However, increased code size in comparison to Java bytecode can hardly be avoided because Java bytecode is a very compact ISA by design.

7. Hardware requirements should be moderate. Minimizing hardware requirements to an absolute minimum is not an objective because RISC pipelines are clearly superior with respect to this characteristic. Complex FU operations should be supported to provide fast execution of programs. On the other hand, complex and energy-consuming techniques for dynamic scheduling or operand matching should be avoided.

8. A prototype of the new processor should be able to run on the same FPGA with the same clock frequency as the bytecode-based processor.

5. Approaching the Solution

After having analyzed weaknesses of the previous bytecode-based AMIDAR processor and having defined research objectives, ideas for achieving these objectives must be collected. Developing a new ISA opens a large field of different approaches. This chapter presents the way to the final concepts of ISA and of operand matching mechanism shortly.

5.1. Changing the ISA

The major design goal for the new ISA is to reduce unnecessary FU operations and data transmissions. On the other hand, operations should be kept on a high abstraction level to facilitate synthesis of CGRA contexts. This can be achieved by removing the operand stack as means for passing data between instructions. Hence, this opens the basic question what other means to use instead.

5.1.1. Register-Based Architecture

An obvious replacement for the operand stack is a register file. The Dalvik Runtime and its successor, the Android Runtime [14], define such an architecture. Each register can be treated independently of other registers while a stack must always be treated as a whole. This simplifies reconstructing data dependencies and reordering instructions. Thus, register-based architectures help to achieve a higher degree of concurrency compared to stack-based architectures. However, if the architectural register file was implemented simply as a separate FU, the same problems would persist. The FU would become a bottleneck because it would serve as data source and sink for most instructions. The number of operations and data transfers would not be reduced. This problem can be solved only if architectural and physical register files are treated as separate entities. Then, out-of-order processing techniques can be applied (see section 3.3.7). Unfortunately, these techniques increase hardware complexity severely. Consequently, it seems worth looking at other alternatives.

5.1.2. Moving Closer to Microarchitecture

The most promising way to avoid a bottleneck in instruction processing without leveraging expensive out-of-order techniques is to avoid a central storage element for passing data. Instead, the FU structure of the processor and transmissions between FUs must be visible to the programmer directly. This brings the ISA closer to microarchitecture by making the processor programmable on token level. TTAs apply a similar design principle (see section 3.4). They provide almost full access to microarchitecture details. However, the TTA approach is no ideal choice for AMIDAR because it leads to large code size, forbids runtime reconfiguration, and makes CGRA context synthesis more challenging.

Another technique for specifying data transfers is required. AMIDAR matches operands with FU operations dynamically, similar to data flow architectures. This fosters the idea to apply data flow concepts at the ISA level as well. Thorough study of data flow architectures (see section 3.3) reveals that pure data flow architectures come with severe disadvantages. Operand/operation matching is complex, data replication is inefficient, and sequential memory semantics can be realized with high effort only. However, it is possible to stick with control flow based issuing of instructions while data transfers are specified using data flow techniques. This compromise avoids the aforementioned problems of pure data flow architectures.

Data flow architectures basically use two alternative techniques for specifying data flow between instructions. The first one is to use static code memory addresses of target instructions (Manchester Machine, section 3.3.2). The second one is to use frame addresses (Monsoon, section 3.3.3). A frame is allocated for each function call or loop iteration. Operands are matched at static addresses within a frame. Unfortunately, managing frames is not an easy task. The second approach would require a complicated frame store to be integrated into each FU. Consequently, the first approach is preferred for AMIDAR. Addresses relative to the sending instruction can be used instead of absolute addresses in order to improve compactness of code. Because the new ISA uses control flow for issuing instructions, target addresses can be specified based on the dynamic issue order imposed by control flow. This avoids explicit switch instructions, which send data to alternative targets depending on decision values. Pure data flow architectures can use only static code memory addresses because they do not have a predictable sequence of instruction execution.

5.1.3. Final Concept

This section explains the generic concept of the new ISA. It has been developed based on the considerations of previous sections and on the evaluation process which will be covered by chapter 11. Details of the ISA will be presented in chapter 6. The new architecture borrows ideas from data flow architectures for specifying data transfers between instructions. Instruction semantics are kept close to Java bytecode. Therefore, the new architecture will be referred to as *Data Flow Oriented Java Architecture (DOJA)* from here on.

Instruction

The assembler representation of an instruction is depicted in fig. 5.1. It consists of four components. Every instruction contains an operation code. Some operations require an additional constant, e.g. for immediate values or branch targets. If an instruction produces a result, the *result target reference* defines which instruction will receive this result. It consists of an *instruction offset* and a *port*. The offset counts the number of instructions to execute after the current instruction until the result is received. A value of 0 references the next instruction. The port allows to distinguish between multiple operands of the target instruction. [9]

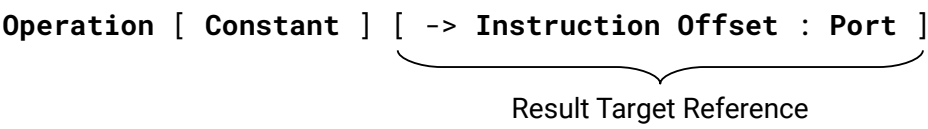


Figure 5.1.: DOJA instruction in assembler representation [9]

Example

Figure 5.2 shows the DOJA instructions which are generated from the bytecode of the running example (listing 1, fig. 2.3). Control and data flow between instructions is illustrated as well. `invoke_fs_i 5` in line 3 creates a stack frame with 5 local variable (LV) slots for this method. It also receives the address of the instruction at which to continue after returning from the method. The stack of method frames for storing LVs is retained whereas the operand stack has been removed. More details on calling convention will be given in section 6.4. Line 1 creates the constant 1 and sends it to instruction offset 2 at port 0, which is `getfieldh` in line 4. Line 2 creates the constant 103 and sends it to the same instruction at port 1. `getfieldh` treats the value received at port 0 as handle and the value at port 1 as offset. It loads the corresponding field `arrayA` and sends its value to the next instruction. `write 2` writes this value to LV 2 (variable `a`). Lines 6 to 9 repeat the same procedure to write the value of field `arrayB` to LV 3 (variable `b`). Lines 10 to 12 read the handle of array `a`, obtain its length, subtract 1, and send the result as starting value of `i` into the loop. Line 13 sends 0 as starting value of `sum` into the loop.

The loop reaches from line 15 to line 30. `writer` (write with result) in line 16 receives the starting value of `i` and writes it to LV 4. In addition, it sends this value to the `brl` instruction. The stop condition of the loop is checked here. Branching works like in other control flow based architectures. If `i` is less than 0, control flow continues at line 32. Otherwise, control flow continues at line 19. `fwd` in line 15 receives the starting value of `sum`. Depending on the branch decision, this value is forwarded either to the next `fwd` in line 22 or to `push32` in line 35.

Lines 19 to 21 as well as lines 23 to 25 read the next elements from arrays `a` and `b`. The elements are multiplied in line 26. The result is transferred to `add` in the next line. The second operand of this addition is received from `fwd` in line 22. The new sum is sent to `fwd` in line 15. Consequently, intermediate values of `sum` are passed directly from one iteration of the loop to the next one without intermediate storage. Lines 28 and 29 decrement `i` by 1 and send it to `writer 4` in line 16. `goto` executes an unconditional jump to the beginning of the loop.

After the loop has been left, invocation of the `println` method is prepared. Method arguments are passed using the stack of method frames. `push` instructions store arguments directly on top of the current frame. They will be accessible as LVs in the invoked method. Lines 32 and 33 load the output stream handle from the static field `System.out`. The offset of the field is inlined into the `getfieldh_i 16` instruction. Inlining is possible here because the offset is small enough to fit into the instruction. The "+" after the instruction indicates that the handle is duplicated. This duplicate will be picked up by the next `send_again`. Line 34 pushes the output stream handle as first method argument. Line 35 pushes the final value of `sum`, which is received directly from the loop, as second argument. `send_again` in line 36 sends the handle duplicate generated in line 33 to `getcti`, which determines the class table index (CTI) of the output stream object. The CTI identifies the

```
1    imm           1 -> 2:0
2    imm         103 -> 1:1
3    invoke_fs_i   5
4    getfieldh       -> 0:0
5    write         2
6    imm           1 -> 1:0
7    imm         104 -> 0:1
8    getfieldh       -> 0:0
9    write         3
10   read          2 -> 0:0
11   getsize         -> 0:0
12   add_i        -1 -> 2:0
13   imm           0 -> 0:0
14 L1:
15   fwd             -> 5:0
16   writer        4 -> 0:0
17   brl          L2
18
19   read          2 -> 1:0
20   read          4 -> 0:1
21   getarray        -> 4:0
22   fwd             -> 4:0
23   read          3 -> 1:0
24   read          4 -> 0:1
25   getarray        -> 0:1
26   mul             -> 0:1
27   add             -> 3:0
28   read          4 -> 0:0
29   add_i        -1 -> 2:0
30   goto         L1
31 L2:
32   imm           1 -> 0:0
33   getfieldh_i  16 -> 0:0 +
34   push32
35   push32
36   send_again      -> 0:0
37   getcti          -> 0:0
38   inv_virt_i   13 -> 2:0
39
40
41
42   return_fs_i   1 -> 0:0
43   return_tm
```

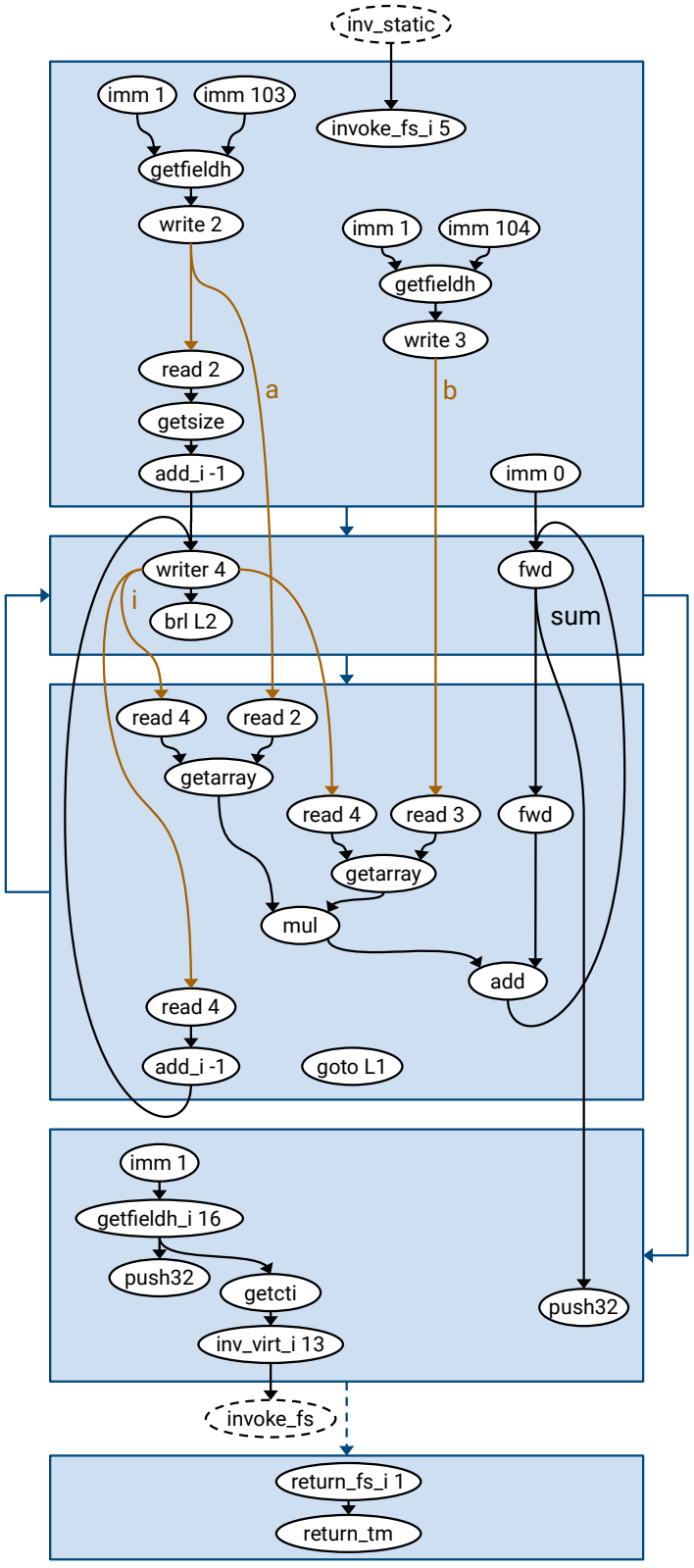


Figure 5.2.: DOJA instructions for the running example

runtime class of the object. It is transferred to `inv_virt_i 13` for virtual invocation of the `println` method. This instruction sends the code address of line 42 to `invoke_fs` of the invoked method. After returning from the `println` method, the example method also returns using the instructions from lines 42 and 43. `return_fs_i 1` removes the method frame and sends the return address to the next instruction. `return_tm` jumps to this address in the calling method.

Formal Characteristics

The preceding example shows some important features of this kind of data flow description [8].

- Every value which is sent by an instruction must have exactly one receiver on every possible path of the program.
- Every executed instruction must receive exactly one value at each of its ports on every possible path of the program.
- Each data transfer originates from exactly one instruction and ends at exactly one instruction for each possible path of the program.

This conforms to the single assignment property of traditional data flow architectures. Every port of an instruction can behave similar to a Φ function as known from SSA forms in compiler engineering. Port 0 of `writer` in line 16 is an example. The result of either the instruction in line 12 or the instruction in line 29 is received depending on the previously executed program path. In contrast to Φ functions, the input of a port does not depend only on the immediately preceding basic block. Theoretically, the input can depend on the full preceding control flow path.

Similarly, the result of an instruction can have multiple alternative targets depending on the future control flow path. The result of `fwd` in line 15 serves as example. It is either sent to the instruction in line 22 or to the instruction in line 35. Since each instruction can specify only one instruction offset, these targets must have the same position relative to the source instruction in issue order. This is called the *multi-target problem*. In the example, the problem is solved by inserting the second `fwd` in line 22. The first `fwd` in line 15 could be removed. It is the effect of a weakness in the code generation process.

Execution Characteristics

Figure 5.3 demonstrates how one iteration of the loop of the running example is executed by the processor. Again, latencies for operation execution and data transmission are ignored. Every DOJA instruction corresponds to one token with one FU operation. The FU operations of the last line already belong to the next iteration. In comparison to the corresponding Java bytecode example of fig. 4.1, all stack operations are removed. In addition, no `imm` operations are required to generate constants on the Token Machine. Small LV addresses and the constant operand of the second addition are sent to the corresponding FUs along with the operation codes over the TDN. Furthermore, data is transmitted between Heap, Integer Multiplier, and Integer ALU directly. Hence, the number of data transfers is reduced. More concurrent processing of FU operations is enabled as well. For example, the handle of the second array is obtained from the Frame Stack while

the first array is being accessed on the Heap. In consequence, all these characteristics lead to a much shorter execution sequence.

```
L1:
  fwd        -> 5:0
  writer  4  -> 0:0
  brl     L2
  read    2  -> 1:0
  read    4  -> 0:1
  getarray   -> 4:0
  fwd        -> 4:0
  read    3  -> 1:0
  read    4  -> 0:1
  getarray   -> 0:1
  mul        -> 0:1
  add        -> 3:0
  read    4  -> 0:0
  add_i  -1  -> 2:0
  goto    L1
L2:
```

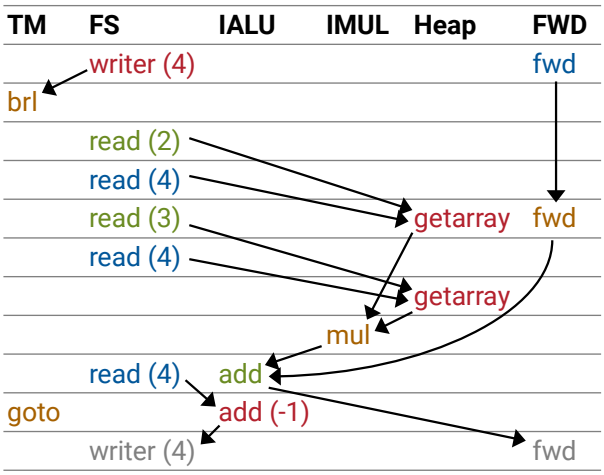


Figure 5.3.: Execution of the loop body of the running example on the DOJA-based processor

5.2. Operand Matching

An essential feature of AMIDAR is the independence of FUs with respect to their timing. Consequently, a mechanism is required for matching operands received over the data interconnect with FU operations. As discussed in section 4.1, the current tag-based mechanism exhibits performance problems because of long combinatorial paths and limited concurrency. Hence, alternative mechanisms should be investigated. Before looking at these alternatives, it is important to analyze the matching problem exactly. Figure 5.4 shows an excerpt from the DOJA code for the running example. The first `push32` pushes the output stream handle which has been loaded from heap memory onto the method argument stack. The second `push32` pushes the integer value to print. The `push32` instructions have an implicit state dependency between each other to ensure the correct order of method arguments. However, the second argument will most likely arrive earlier at the Frame Stack than the first argument because heap accesses are much slower than data forwarding. If no mechanism for matching operands with their corresponding operations existed, method arguments would be ordered wrongly. In general, such an operand conflict can occur whenever two distinct FUs send data to identical target FU and port. Some conflicts might be resolved on software level by data or control dependencies between instructions, e.g. if the `fwd` received data from the first `push32`. The values 145 and 68 in fig. 5.4 are chosen arbitrarily to exemplify the operand matching mechanisms in the following sections.

The Token Machine of the bytecode-based processor sends operations together with target information to FUs. However, the result target of a DOJA instruction is not known yet when decoding the source instruction. Consequently, the Token Machine of a DOJA-based processor sends FU operation codes before corresponding target information. Operation

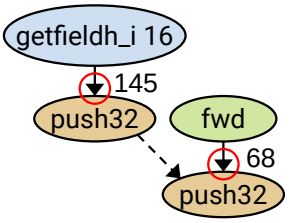


Figure 5.4.: Operand conflict in the running example. Colors of instructions indicate the executing FUs.

codes are sent immediately when instructions are decoded whereas target information is sent after target instructions have been decoded. This must be considered for all operand matching mechanisms.

5.2.1. Tags

The tag-based matching mechanism has already been explained in detail in section 2.2.1. Figure 5.5 illustrates it on a more abstract level applied to the example from fig. 5.4. Both Heap and Forwarding Unit have stored the results into their result queues. Corresponding target information is located in parallel target queues. The FUs try to send the results to the Frame Stack simultaneously. Only the Heap value is accepted because its tag matches the tag at the front of the operation queue. The Forwarding Unit value is rejected for now. It will be accepted after the first operation is consumed by the Frame Stack.

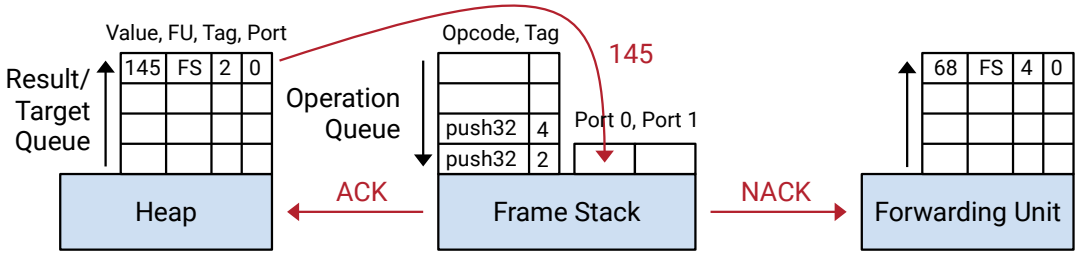


Figure 5.5.: Operand matching with tags

5.2.2. Operation Memories

The next mechanism has similarities to the explicit token store of Monsoon (see section 3.3.3). FUs have multiple storage slots per port. The addresses of these slots are used for matching operands. Monsoon allocates these slots statically. However, this requires a memory which is large enough to guarantee that no address is used by multiple instructions simultaneously. Furthermore, a costly separation in memory frames is required. This would be overdesigned for FUs of AMIDAR processors. Instead, slots are allocated dynamically.

This mechanism is depicted in fig. 5.6. Parts of the following description have already been published in [8]. The operation queue and the operand storage slots of an FU have been combined to an *operation memory*. Each line of an operation memory stores one operation together with its operands. The address of a line is called *operation address*.

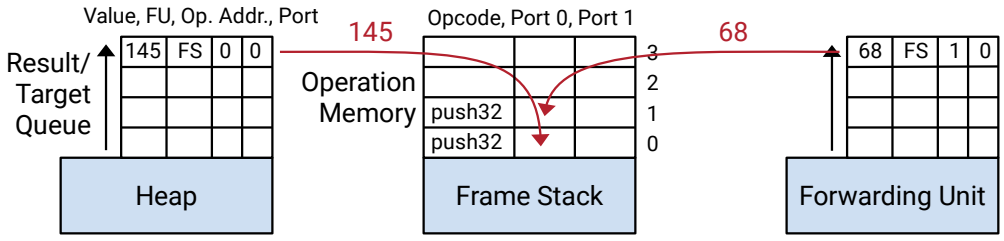


Figure 5.6.: Operand matching with operation memories

Operations are written and read cyclically. Before the next operation can be written to a line, this line must be read and sent to execution. An operation can be sent to execution only if all its operands have been stored into the memory. Operation storage has FIFO semantics. Hence, instructions which are mapped to the same FU are executed in the order they have been decoded. In contrast, operands can be stored to the memory in any order using operation address and port. In the given example, the Forwarding Unit can send its result to the Frame Stack before the Heap without interchanging method arguments. Decoding is blocked if a target queue is full or no free operation address is available. This will prevent further target information to be delivered by the Token Machine. Since no data is transmitted without target information, data is never sent to an FU without corresponding free space in operation memory. Consequently, no acknowledgment signal is required from the receiver to the sender.

5.2.3. Instruction Decoder Stops

The third proposal for matching operands with operations [77] aims at reducing required hardware resources. In contrast to previous alternatives, it is assisted by the compiler and must be considered in ISA design. The compiler or programmer identifies all potential operand conflicts and marks them in instructions. The first `push32` must be marked critical at port 0 for example. Unfortunately, assignment of operations to FUs must be fixed at compile time for the detection of operand conflicts to be feasible.

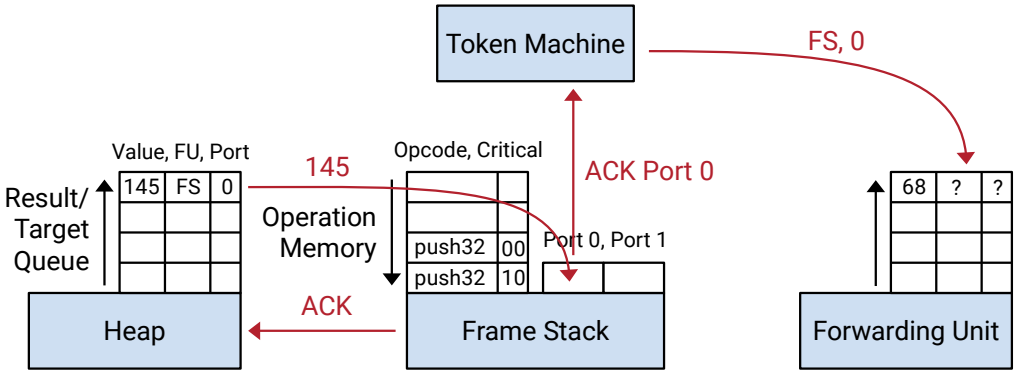


Figure 5.7.: Operand matching by stopping the instruction decoder

The hardware part of the mechanism is illustrated in fig. 5.7. The criticality flag causes the Token Machine to stop decoding the next instruction which uses the same port on the same FU. In the example, decoding is stopped immediately after the first `push32`. Since

the second `push32` is not decoded, the Forwarding Unit does not receive target information for its result value. Consequently, it does not send this value. The criticality flag is also transferred to the Frame Stack along with the operation code. As soon as the Heap sends its result value to the Frame Stack, this flag causes an acknowledgment to be sent to the Token Machine. It enables decoding again, which causes missing target information to be delivered to the Forwarding Unit. In some situations, the acknowledgment can be sent to the Token Machine before decoding is stopped. No stopping is required in these cases. The acknowledgment from receiving to sending FU avoids overwriting data in port registers if this data is not accepted by the FU immediately due to internal delays.

This mechanism moves most synchronization complexity to the Token Machine. The red signal path in fig. 5.7 is hard to realize by purely combinatorial logic because it spans over many FUs. At least one register can be assumed to be inserted into this path.

5.2.4. Discussion

The performance of the tag-based mechanism suffers from limited throughput because of the critical acknowledgment path. In addition, it limits concurrency because operands are accepted for the oldest waiting operation only. The mechanism based on operation memories eliminates direct acknowledgments between FUs and accepts operands for all waiting operations. This makes fair arbitration of incoming requests superfluous as well. A simpler arbitration scheme with fixed priorities is sufficient. Therefore, a higher performance can be assumed here. The technique based on decoder stops removes tag comparison from the acknowledgment path. However, it introduces another long acknowledgment path over the Token Machine. Since the whole instruction decoding process is stopped to enforce correct ordering of operands, the overall performance can be expected to be lower than with tags.

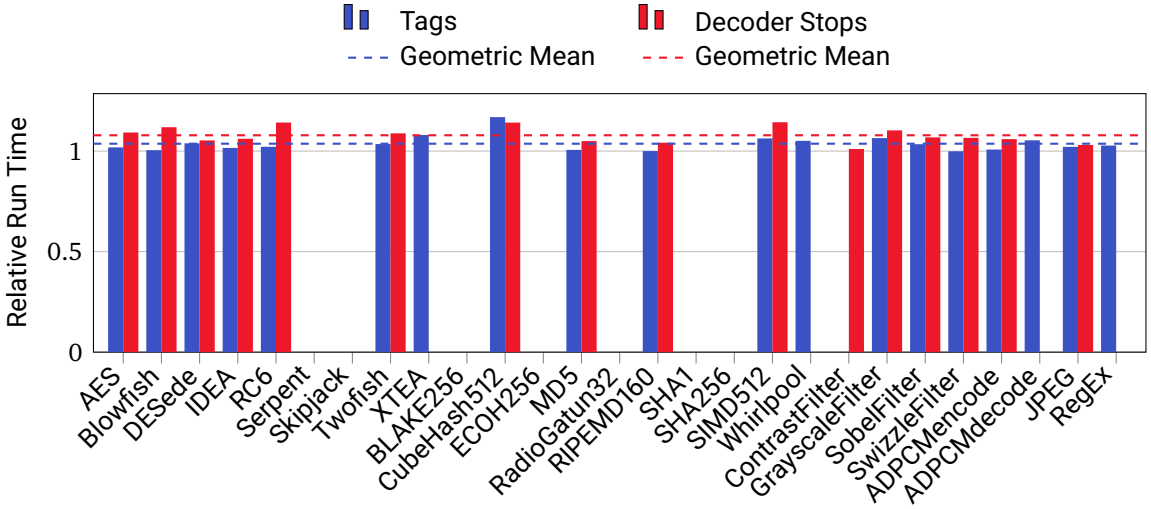


Figure 5.8.: Simulated benchmark run times for different operand matching mechanisms relative to the variant with operation memories

Several benchmark programs (see section 10.1) are converted into DOJA code and are executed in a software simulator with different matching mechanisms. This simulator models processor components in an abstract way. Therefore, it provides high flexibility

but timing is not precise. Especially clock frequency is not taken into account. Hence, the following numbers must be seen more as estimations than as exact measurements. The benchmark run times from fig. 5.8 confirm the performance assumptions. However, the difference between these techniques is lower than expected. Tags increase run times by 3.7 % on average compared to operation memories. Decoder stops increase them by 7,9 %. Missing bars indicate that compilation or simulation of the final code fails for the corresponding benchmarks. Code conversion succeeds for all benchmarks with operation memories. Instruction scheduling fails for 8 benchmarks with tags and decoder stops. Simulation fails for additional 3 benchmarks with the mechanism based on decoder stops. Failed simulation indicates an unsolved error in simulation or code conversion. Consequently, another important conclusion can be drawn from this experiment: Instruction scheduling is more complicated with tags or decoder stops than with operation memories because of less scheduling freedom. Marking critical instructions increases compiling effort further for the variant with decoder stops.

Hardware effort can be estimated only roughly without implementing all mechanisms in hardware. The mechanism based on tags requires tag counters and tag comparators. The mechanism based on operation memories requires operation address counters and additional memory for operands. Operation memories can have small depths. Therefore, they can be implemented efficiently with distributed RAM on FPGAs. The mechanism based on decoder stops requires transmission and storage of criticality flags only. Consequently, hardware effort is expected to be highest for operation memories and lowest for decoder stops.

	Performance	**Hardware Effort**	**Compiling Effort**
Decoder Stops	🔴	🟢	🔴
Tags	🟡	🟡	🟡
Operation Memories	🟢	🔴	🟢

Table 5.1.: Comparison of operand matching techniques

Table 5.1 compares the strengths and weaknesses of the presented operand matching techniques. Performance has been defined to have higher priority than hardware effort in section 4.2. Hence, operation memories are preferred. Lower compiling effort is the most important argument in favor of operation memories. Instruction scheduling turns out to be a difficult challenge even with this mechanism (see section 9.5). Therefore, this technique is chosen for hardware implementation and further research.

Part III.

Implementation

6. Instruction Set Architecture

The general concept of the new ISA named *Data Flow Oriented Java Architecture (DOJA)* has already been presented in section 5.1.3. This chapter explains important details of the architecture as well as the binary representation of programs. A full instruction set listing can be found in appendix A.

6.1. Replicating Data

Efficient data replication is a hard challenge with pure data flow architectures. Since DOJA is based on control flow, memory can be used for this purpose. Two types of memory have been inherited from Java bytecode: heap memory and LV slots in stack frames. Scratch pad memory (SPM) has been added as a third type. It is designed as faster but smaller alternative to LVs. Similar to a register file, SPM provides a small amount of storage slots which are not organized in frames. Hence, its contents must be saved on method calls. However, evaluation has shown that integration of scratch pad functionality into the Frame Stack is advantageous (see section 11.3). Consequently, `read` and `write` instructions access LVs. They differ to `load` and `store` instructions only by specifying LV addresses directly inside instructions instead of utilizing separate instructions for address specification. A minimalist example is provided in fig. 6.1a. LV slot 4 is used for replication. `write` just writes a value to LV storage while `writer` (write with result) also forwards the written value to another instruction directly.

(a) By local variable slots

(b) At FU output

Figure 6.1.: Techniques for data replication

Another technique, which is closer to the data flow principle, is shown in fig. 6.1b. The *keep result* flag of `add_i 1` (indicated by "+" in assembler code) causes the result to be duplicated. This is realized by transmitting the result without removing it from the result queue. `send_again` picks up the duplicate and transfers it to another instruction. Although `send_again` does not need to follow the duplication immediately, there are some limitations to instruction scheduling:

- No other instruction which can be executed on the same FU as the duplicating instruction may be placed between duplication and `send_again`.

- A duplicate must be picked up by `send_again` before the next duplication.

Direct duplication at FU outputs is preferred for low fan-outs and for targets which are close to the duplicating instruction, especially within basic blocks. Otherwise, LV storage is preferred. Both mechanisms can be found in the running example (fig. 5.2).

6.2. Discarding Data

With register-based architectures, values are discarded implicitly by overwriting registers. With DOJA, a value which is sent by an instruction must be received by another instruction on every path of the program (see section 5.1.3). Therefore, it is necessary to discard values explicitly if they are not required in a branch of the program. The `disc` operation can be used for this purpose. An example is shown in fig. 6.2. The value stored in LV 2 is compared to zero. If it is greater or equal zero, the value is forwarded. Otherwise, it is discarded and replaced by zero. `disc` is identical to `nop` in binary encoding. These operations are distinguished only during code generation to detect errors as early as possible. `disc` or `nop` are the only operations which can receive an arbitrary number of values, including no values. Discarding is realized by removing data words from result queues without sending them.

```
1    read  2 -> 1:0
2    read  2 -> 1:0
3    brl  L1
4    fwd     -> 1:0
5    goto L2
6  L1:
7    disc
8    imm   0 -> 0:0
9  L2:
10   write 2
```

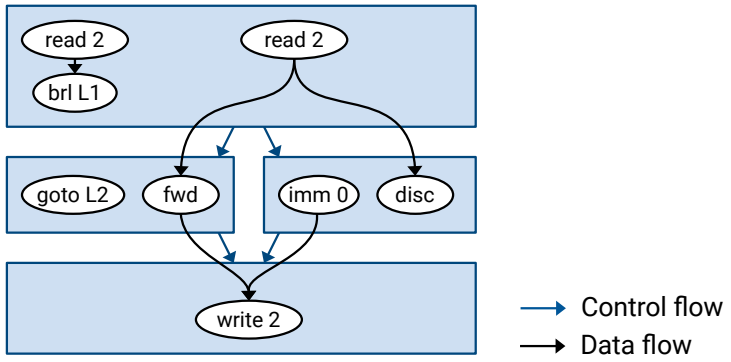


Figure 6.2.: Example for discarding data

6.3. Multi-Target Problem

The multi-target problem has already been discussed roughly in section 5.1.3 with respect to the running example. The result of an instruction can have multiple alternative targets depending on the future control flow path. However, each instruction contains only one result target reference composed of one instruction offset and one port. Consequently, all targets must have the same position relative to the sending instruction in issue order. This problem can be solved by inserting a `fwd` as in the running example. Figure 6.3 illustrates another solution. `read 2` in line 2 has two targets. A `nop` has been inserted before `write 2` to achieve a common instruction offset. Useful instructions can often be preponed instead of inserting `nop` instructions.

A similar problem arises with the port specification. `sub` expects the value at port 1 whereas `write 2` expects it at port 0. Furthermore, subtraction operands cannot be

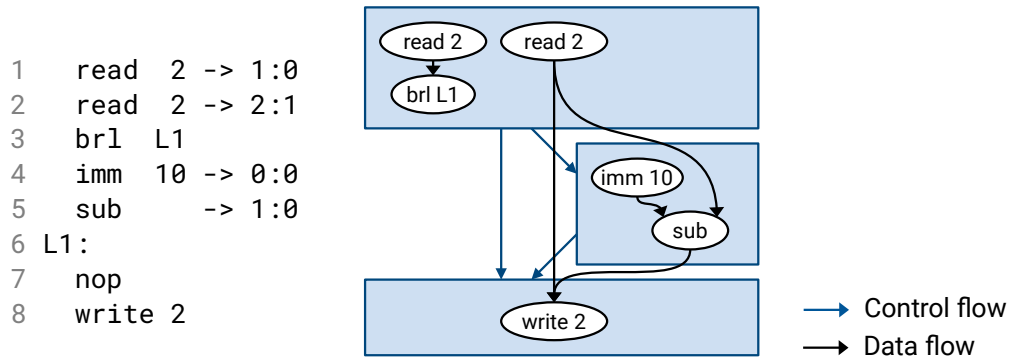


Figure 6.3.: Example for an instruction with multiple targets for its result

exchanged. The solution is simple here: The port specified by the source instruction is ignored if the target instruction receives operands only at port 0. Such instructions, like `write 2`, redirect incoming data to port 0 automatically. A `fwd` can be inserted in other situations. This technique could be generalized to all operations which have a single operand. However, all operations which expect their single operands at port 1 use it for receiving constants. Constants never run into the multi-target problem because they do not need to be transferred across basic block boundaries. Consequently, this extension would not gain any advantages.

6.4. Method Calls

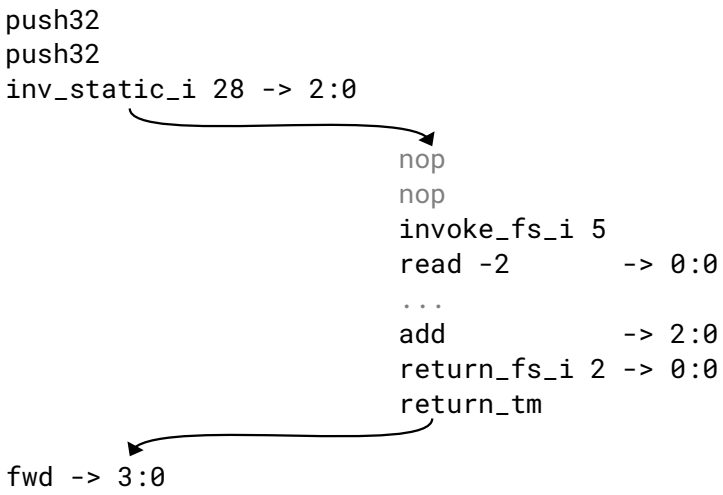


Figure 6.4.: Invocation of a static method with 2 arguments, 5 LVs, and return value

Figure 6.4 gives an example how a static method with 2 arguments, 5 LVs, and return value is called. Method arguments are prepared by `push` instructions. `inv_static_i 28` resolves the start address of the method with absolute method table index (AMTI) 28 and jumps to this address. Tables will be explained in section 6.7. The code address of `fwd` is sent as return address to instruction offset 2. This offset is defined as part of the calling convention. `invoke_fs_i 5` is placed at this position. It creates a new stack frame with 5 LV slots and receives the return address. The first two instructions of a method can

be used for generating the number of required LV slots if it does not fit directly into the instruction. Otherwise, they can be used for arbitrary instructions which do not depend on stack frames. Method arguments can be loaded from negative LV slots after stack frame creation (last argument in slot -1). In this example, `add` produces the return value of the method, which must be sent to port 0 of the first instruction after return. The stack frame is discarded by `return_fs_i 2` where 2 specifies the number of method arguments. The return address is delivered to `return_tm`, which jumps to this address. Finally, the return value is received by `fwd`.

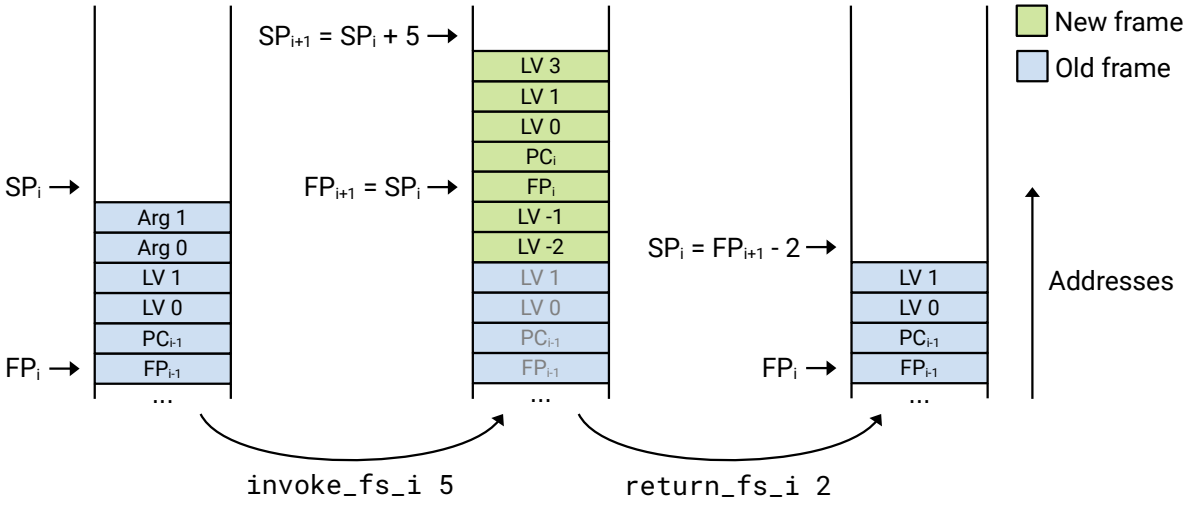


Figure 6.5.: Frame Stack memory layout

The Frame Stack memory layout for this method call is depicted in fig. 6.5. The active frame is defined by the frame pointer (FP). The next free address is defined by the stack pointer (SP). All LVs are accessed relative to the FP. Method arguments are stored at negative LV addresses. Addresses 0 and 1 are reserved for the previous FP and the return address (PC). Higher addresses can be used for normal LVs. Before a method is called, arguments are pushed on top of the LVs of the current frame. Invocation stores old FP and return address into memory. Then, pointers are updated. Afterwards, method arguments lie below the FP. Return discards the top frame by reverting pointers. Each LV slot can store a 32 bit value. 64 bit values consume two successive slots.

6.5. Exception Handling

The bytecode-based AMIDAR processor implements exception handling in hardware. The more fine-grained control of hardware provided by DOJA simplifies exception handling in software. Because it is not considered performance critical, exception handling has been moved to software in order to save hardware resources. The new process is illustrated in fig. 6.6. An exception is thrown by calling a special assembler routine. This routine determines the *throw address*, which is the code address where the routine has been called, and obtains the CTI of the exception object. Both values are passed to a normal Java method. It uses the throw address and the *method meta table* to determine the throwing method. Since entries of the method meta table are ordered by code address, a binary search delivers the result quickly. The corresponding entry in this table holds the exception

table index for the method. Starting at this index, the *exception table* is searched for a handler which covers the throw address and the CTI of the exception. Then, the assembler routine jumps to the start address of the handler. The handle of the exception is sent to port 0 of the first handler instruction. Details about the tables can be found in section 6.7.

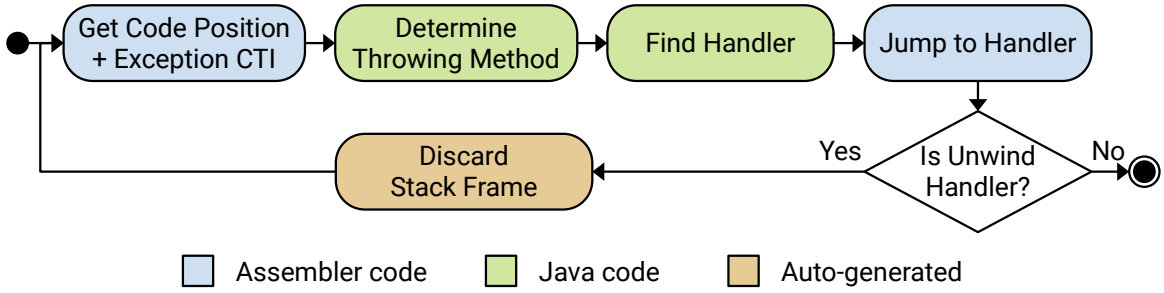


Figure 6.6.: Exception handling process

Each method has an additional auto-generated handler which covers the whole method for all exception classes. This universal *unwind handler* is appended to the end of the normal method body. Consequently, handler search can always find a handler. In case the programmer has not specified an exception handler, the unwind handler will be called. It discards the top method frame and calls the assembler routine again in the context of the previous method. Hence, the stack is unwound until a handler created by the programmer is found.

6.6. Binary Instruction Format

This section is based on the description of the binary format as it has been published in [8, 9]. Every instruction has a width of 24 bits. This is the smallest multiple of one byte which can store all relevant information and leaves small room for extensions. Four types of instructions exist as depicted in fig. 6.7. The type is encoded in the two highest bits. Bit 21 is reserved for future extensions.

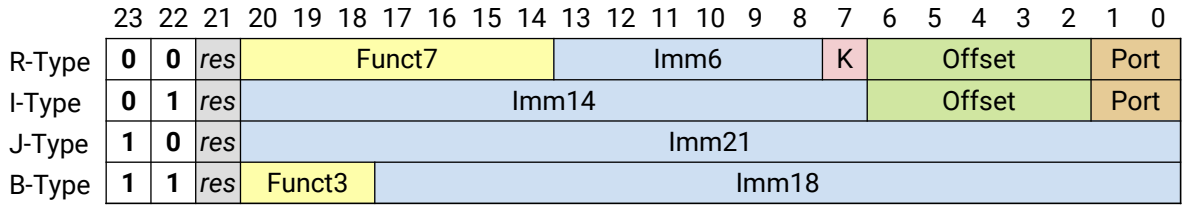


Figure 6.7.: Binary encoding of instructions

- R-type is used for most instructions. The *Funct7* field holds the operation code. Some operations like `read` or `write` use the *Imm6* field as constant operand. Whether the constant is interpreted as signed or unsigned, depends on the operation. Bit 7 is set to 1 if the result should be duplicated as explained in section 6.1. The remaining bits contain instruction offset and port.

- I-type is used for sending constant values. Signed constants up to 14 bit can be stored in the *Imm14* field. They are biased by $6144 = 3 \cdot 2^{11}$ to allow more constants to fit into the field. Larger constants must either be computed or stored in constant pool. Special operations exist for loading these constants from the pool. The remaining bits contain instruction offset and port again.

- J-type is used for unconditional jumps. The *Imm21* field holds the byte address of the jump target relative to the current code position.

- B-type is used for conditional branches. The *Imm18* field holds the relative byte address of the branch target again. The comparison which decides whether the branch is taken or not is encoded in field *Funct3*.

6.7. New AMIDAR Executable Format

The new AMIDAR executable (NAX) format is the binary program image for DOJA-based processors. It contains code, constants, and meta information for managing objects. The boot loader writes it into the external DRAM starting at address 0. Most of the NAX layout has been adopted from its predecessor, the AXT format [15]. Therefore, this section gives only an overview and focuses on improvements.

The Header holds the starting addresses of all other sections of the image.

The GC Info Section is required for exact garbage collection. The major part consists of bit maps to indicate which object fields hold handles.

The Class Table contains one entry for each class of the program. It is used for `instanceof` checks and virtual/interface method invocations. The class table index (CTI) is used as class identifier throughout the system. The AXT format has specified the super class in each class table entry whereas the NAX format orders entries according to inheritance relationships instead: A class is followed by all its direct and indirect child classes. Each table entry specifies the index of the last child class. This simplifies inheritance checks for classes to single CTI comparisons.

The Method Table provides code address resolution for method invocations. The table maps an absolute method table index (AMTI) to the start code address of the method. The AMTI is used directly for calling static methods. Virtual or interface invocations compute the correct AMTI depending on the class of the object on which the method is called.

In the AXT format, this table has also stored meta information for exception handling and stack management. Exception handling information has now been moved to the separate method meta table because exception handler search is done in software. Information for stack management is included in method code as constants provided to `invoke_fs` and `return_fs` instructions.

The Method Meta Table has been introduced newly and contains one entry of exception handling information for each non-abstract method of a program. The method meta table index (MMTI) identifies each non-abstract method uniquely. Entries are ordered by code address, which enables fast search of the corresponding method for a given code address. The structure of an entry is shown in table 6.1. All methods have at least one exception handler, the unwind handler (see section 6.5). Since this is the only handler for most methods, a simplified encoding has been introduced for this case. If the exception table length is 0, the exception table reference directly holds the code address of the unwind handler relative to the beginning of the method code.

Name	Width	Description
Code Reference	32 bit	Absolute start address of the method code in bytes
Exception Table Length	16 bit	Number of entries in the exception table for this method
Exception Table Reference	16 bit	First entry in the exception table for this method

Table 6.1.: Structure of a method meta table entry

The Exception Table concatenates the exception tables of all methods which have more exception handlers than just the universal unwind handler. Each entry of the table specifies one exception handler.

The Implemented Interfaces Table holds bit strings to indicate which interfaces are implemented by a class or which interfaces are extended by another interface.

The Interface Table is used for calculating the AMTI for an interface invocation.

The Constant Pool stores all constant numbers or handles which do not fit into the binary encoding of instructions.

The Code Section concatenates the instruction streams of all non-abstract methods.

The Immortal Heap holds constant objects such as string constants and class objects with reflection information. In addition, statically initialized arrays with primitive elements are prepared here.

The Handle Table maps handles to physical addresses of objects in immortal or dynamic heap memory. Furthermore, it stores object CTIs and management information for garbage collection. The handle table of a NAX image is filled with handles of objects on immortal heap. Handles of objects allocated at runtime are appended to the table after the image has been loaded into memory.

6.8. CGRA Interface

The interfaces of the CGRA FU remain mostly unchanged. Contexts and internal tables are configured via Wishbone interface. Live-in data is transferred to the CGRA using `cgra_push32` and `cgra_push64` operations. `cgra_pull32` and `cgra_pull64` transfer live-out data from the CGRA. The 64 bit variants have been introduced newly.

The bytecode-based processor has driven the transfer of live-in and live-out values by a finite state machine (FSM) in the Token Machine. Special block RAM tables have supplied the FSM with necessary information. As DOJA allows more fine-grain control of hardware, this is no longer required. Transfers can be programmed fully in software. In consequence, code patches are larger. A global range at the end of the code section is reserved for this purpose. A part of each code patch can be written to this range. Furthermore, not all live-in or live-out values reside in LV storage. The code patch must handle direct data transfers across loop boundaries as well. For example, if an instruction before the loop sends a value to an instruction inside the loop to patch, this live-in value must be redirected to the CGRA.

The code patch for the running example is illustrated in fig. 6.8. The left side shows the original code where the colored ranges have been replaced by new instructions. The loop which is now executed by the CGRA starts at line 15 and ends at line 29 (inclusive). The right side shows instructions which have been placed at the end of the code section. The starting values of `sum` and `i` are sent from outside the loop to the first two instructions of the loop. These direct live-in values are caught by the `push32` instructions in lines 15 and 16. They are stored onto the method argument stack temporarily. It does not matter to which port the live-in values are sent because these instructions redirect all their operands to port 0. (All operations which receive an operand only on port 0 do this.) Each of the `push32` instructions can receive a single value only. If multiple live-in values were targeted at the same instruction, code patching would fail. However, this case is unlikely to occur at loop entries. None of the tested benchmarks shows this problem. The instructions from line 17 to line 19 jump to an absolute code address at the end of the code section (line 40). `return_tm` is used for this purpose. The address is stored in constant pool. `imm 65` delivers the constant pool index where the jump address has been stored and `const32` loads the address from the pool.

The next instructions in line 40 and 41 initialize the CGRA for the desired kernel. Lines 42 to 53 transfer all live-in values to the CGRA. The values stored on the method argument stack are transferred first, LV values second. This is currently the only use case for `pop` instructions. The handles of arrays `a` and `b` are stored twice in different locations of the CGRA register files. Consequently, LVs 2 and 3 are transferred twice. Lines 54 to 57 execute the CGRA kernel. `getfield_i` reads an arbitrary static field and sends it to the CGRA. The actual value does not matter. It only makes sure that all Heap operations have been executed before starting the CGRA kernel. `brne` makes instruction decoding wait until kernel execution has been finished. Lines 59 to 64 transfer live-out values from CGRA to LVs. This is not necessary in this example. However, the synthesis algorithm does not know whether LVs will be used afterwards because it examines loop instructions only. Handle flags (see section 7.6) must be restored for live-out values since the CGRA does not retain them internally. Lines 65 to 67 jump back to the original method code. The last instruction of the loop body has been modified as well. It pulls the final sum from the CGRA as last live-out value and sends it to the `push32` instruction in line 34.

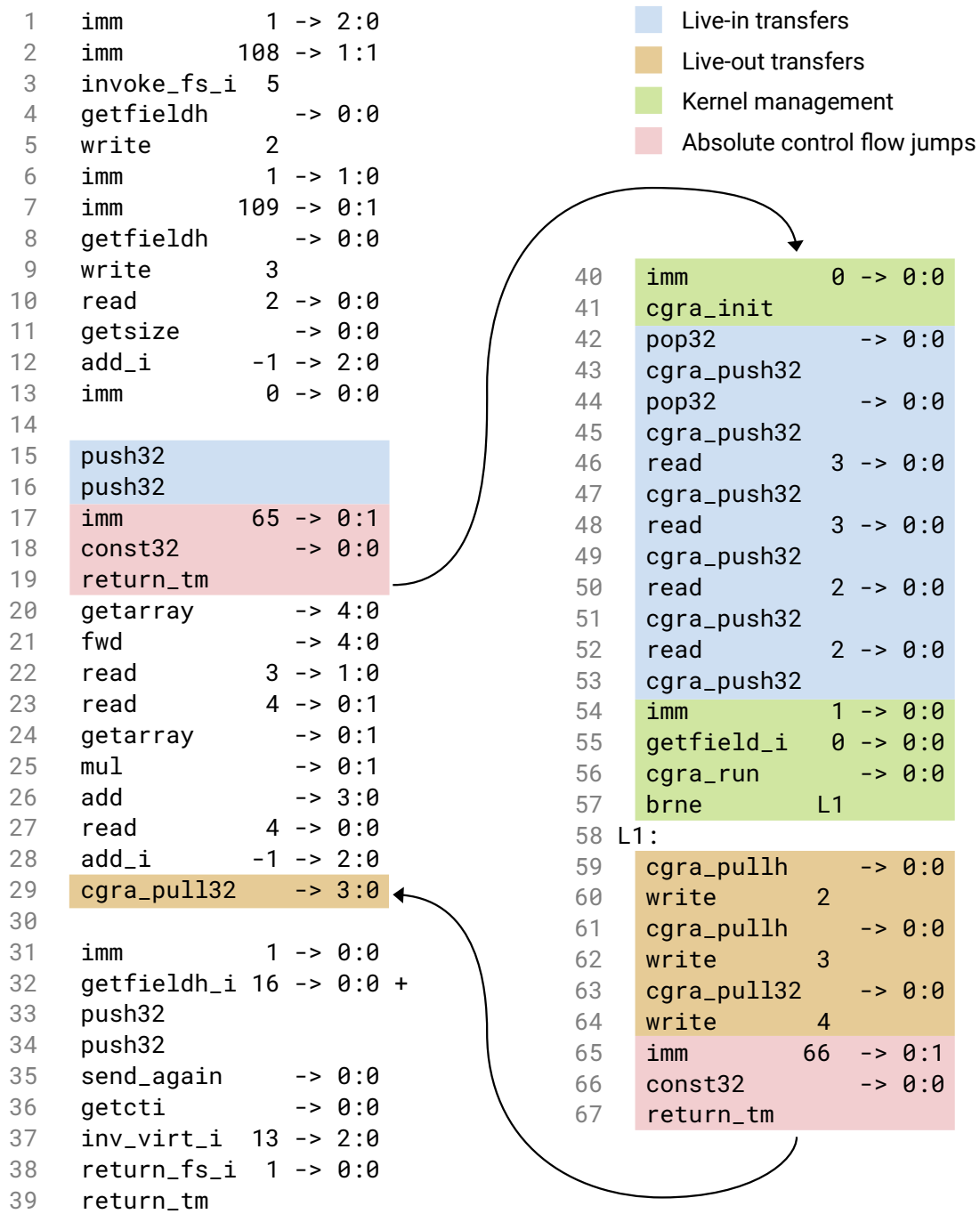


Figure 6.8.: CGRA patch for the running example

Storing directly transferred live-in values on the method argument stack can be avoided if the first two instructions of the loop do not receive live-in values. In this case, kernel initialization can be placed at this position. Live-in values are consumed directly by `cgra_push` instructions afterwards. Unfortunately, this situation rarely appears.

7. Hardware Implementation

Changing the ISA implies changing the hardware implementation as well. However, many parts of the processing system can be kept unchanged due to the modularity of AMIDAR. Figure 7.1 illustrates the hardware architecture of the new DOJA-based AMIDAR system compared to the previous system. The design targets a Xilinx Artix 7 FPGA again. All components outside the processor core remain untouched. ALUs, Thread Scheduler, and CGRA have not been modified apart from interfaces.

The Frame Stack has been re-designed completely because operand stack functionality has been removed. It stores method local data in a stack of method frames as described in section 6.4. It consists of a two-stage pipeline. The first stage computes absolute memory addresses from pointers and FU operands. The second stage accesses memory, which is realized by on-chip RAM blocks. The memory is partitioned into separate stacks for all hardware thread slots statically.

A Forward Unit (FWD) has been added, which consists only of wires from data input to data output apart from control logic. It executes `fwd` operations. Forwarding helps to solve the multi-target problem (see section 6.3) and to avoid deadlocks (see section 9.5.4).

The hardware implementation for multi-dimensional array allocation has been removed from the Heap. Since this process is recursive by nature, it is much easier to implement in software. On the other hand, copying arrays is now supported in hardware. Furthermore, the procedure for collecting the root set of reachable objects for garbage collection has been extended (see section 7.6). The memory subsystem itself has not been changed.

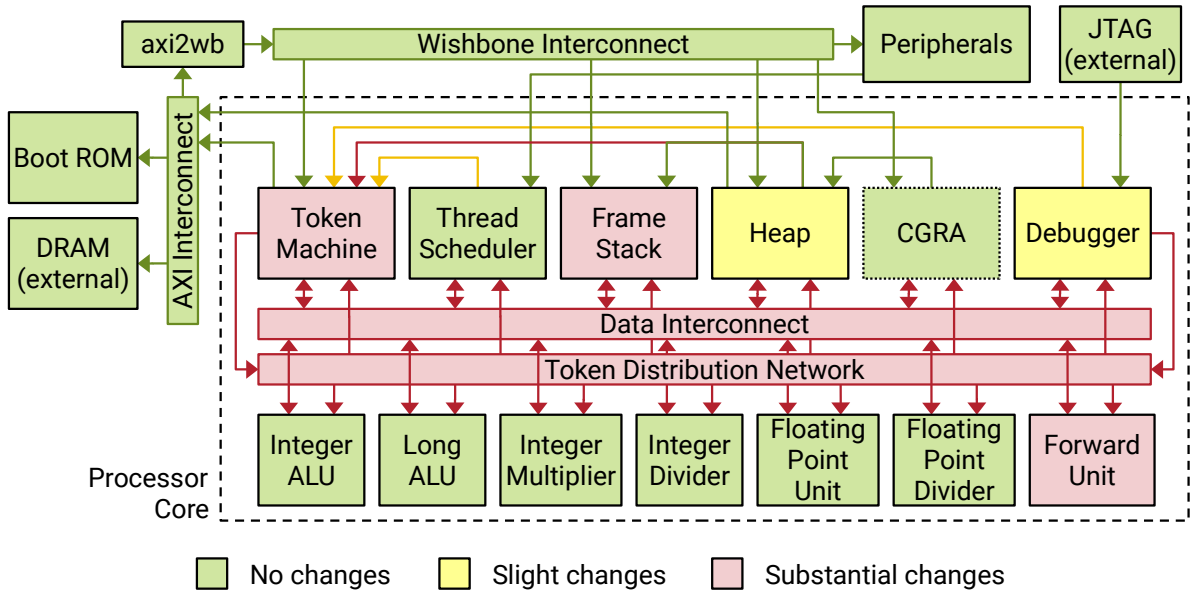


Figure 7.1.: Hardware architecture of the DOJA-based AMIDAR system in comparison to the bytecode-based system

The Debugger had to be modified because of the new operand matching mechanism and a slight change in debugging features of the Token Machine. The host software for controlling the Debugger FU has not been updated yet. Therefore, changes to this FU have not been tested completely. Still, the FU is included in the new system for better comparison. All other changes to hardware will be presented in more detail in subsequent sections of this chapter.

7.1. Token Distribution and Data Interconnect

The generic principle of operand matching with the help of operation memories has been explained in section 5.2.2. This section deals with the hardware components which are required for exchanging data and tokens among FUs in more detail. Tokens are sent to FUs in two parts in order to decouple operation decoding and target resolution. The first part contains the operation code together with an optional, small, constant parameter which originates from the *Imm6* field in R-type instructions (see section 6.6). The second part contains target information (FU, operation address, port). In general, data connections have a width of 64 bit. All signals which are known to transmit only 32 bit data because of limited FU capabilities are reduced to a width of 32 bit automatically.

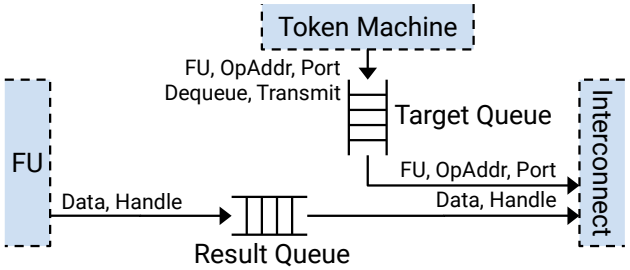


Figure 7.2.: Result storage (without control logic)

Components for result storage are drawn in fig. 7.2. Control logic is not shown for clarity. An FU stores its results into a *result queue* which serves as buffer to the data interconnect. A *handle* flag is carried along with data. It is required for garbage collection and indicates that data is an object handle (see section 7.6). Target information received from the Token Machine is stored into a *target queue*. Results and targets are delivered in the same order. Consequently, the elements at the fronts of both queues belong to each other. Transmission is started if both data and corresponding target are available. The front elements are removed if the data interconnect acknowledges transmission. *Dequeue* and *transmit* flags are delivered from the Token Machine along with target information. The *dequeue* flag indicates whether data should be removed from the result queue on successful transmission. A cleared *dequeue* flag duplicates the result (see section 6.1). The *transmit* flag defines whether a result should be removed from the result queue without transmission. A cleared *transmit* flag discards the result (see section 6.2).

The bytecode-based processor has used direct data connections among FUs. Only a subset of all theoretically possible connections have been realized due to the limited communication patterns of token sets. However, DOJA requires data transfers between arbitrary FUs. Direct connections would result in a full crossbar interconnect. Such a topology would consume a large amount of routing resources and would lead to timing

issues with 12 or more FUs. Consequently, the new processor implements a topology with multiple parallel data buses instead. Each FU can send to all buses but can receive only from one bus [9]. Hence, only a single path exists between a pair of FUs. This restriction simplifies routing logic.

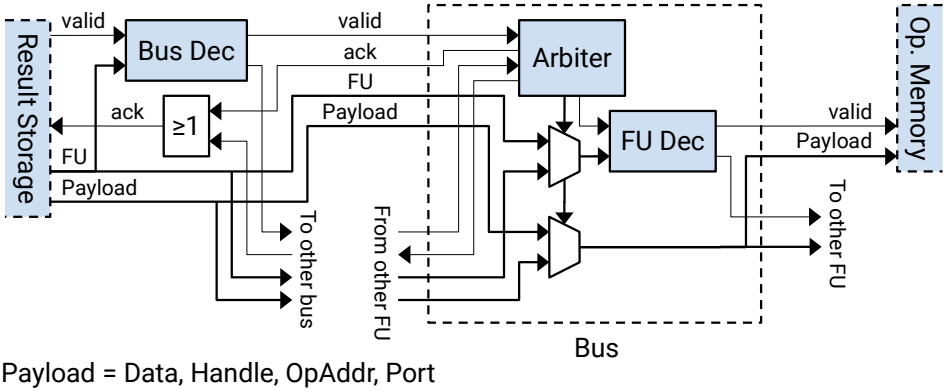


Figure 7.3.: Data interconnect

Figure 7.3 shows a part of the data interconnect. Control signals are visible here because they are essential for interconnect design. Data interconnect payload consists of data, handle flag, target operation address, and target port. The result storage of an FU is connected to a bus decoder. It converts the target FU number into a valid signal for the data bus which provides the path to the target FU. Target FU and payload are routed to all buses. Each bus has an arbiter which receives the valid signals from all bus decoders and selects the request with highest priority. Priorities are hard-coded into the arbiter. Acknowledgment signals are fed back to result storages where they trigger data and target removal from queues. FU number and payload are multiplexed according to arbitration. The FU number is decoded into a valid signal for the corresponding operation memory.

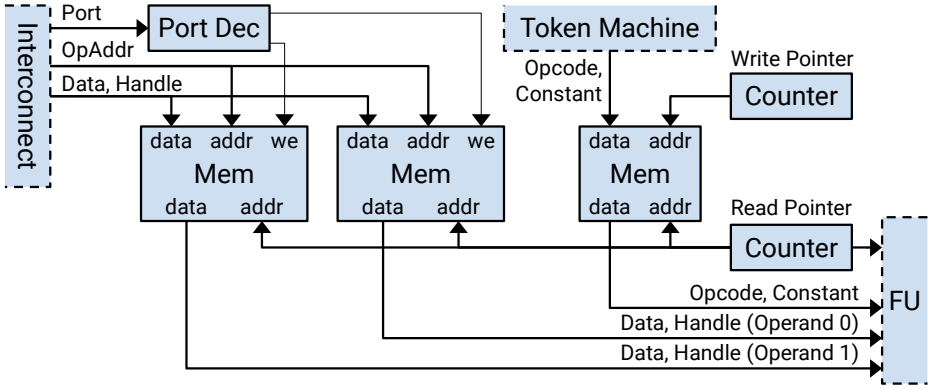


Figure 7.4.: Operation memory (without control logic)

The operation memory of an FU with two ports is illustrated in fig. 7.4. Control logic is hidden again. One memory stores operation codes and constants delivered by the Token Machine. A counter which is incremented on each delivered operation code serves as write pointer. The output of this memory is connected to the FU. A counter which is incremented on each consumed operation serves as read pointer. A valid signal for operation codes is

generated from pointer positions similar to the control logic of a queue. Each port has an additional data memory for storing operands. Data and handle flags are written to these memories directly from data interconnect. Received operation addresses serve as write addresses and received port numbers select the corresponding memory elements. Memory outputs are used as operand inputs of the FU. The operation read pointer provides the read address again. Each entry of a port memory is accompanied by a presence bit which is set on data write and cleared on data read. The FU must check the valid signal of the operation code and the presence bits of corresponding operands. If all operands are available, it can consume the operation. The read pointer value is provided to the FU for identifying the instruction address in case of hardware exceptions (see section 7.4).

All queues and memories are realized using distributed RAM. Queue outputs and FU inputs are registered whereas the data interconnect is purely combinatorial. FU input registers are not shown in fig. 7.4. Memory bypasses exist to achieve the same timing behavior as implied by this schematic. The registers in queues and at FU inputs decouple the combinatorial paths of FUs from the data interconnect and from the rest of the processor.

7.2. Token Generation

The Token Machine is the heart of an AMIDAR processor. It is concerned with multiple tasks:

- Instruction decoding is the main task. Code is fetched from external memory via AXI interface. Decoded FU operations and result targets are transferred to operation memories and result storages of all FUs.
- The Token Machine executes operations like any other FU. It is responsible for control flow changing instructions like branches or method invocations. Because virtual and interface method invocations require access to the class hierarchy, the Token Machine performs *instanceof* checks as well. Additionally, it provides constants to other FUs. Constants can be extracted directly from `imm` instructions or loaded from the constant pool with `const` instructions.
- Context switches are conducted by the Token Machine (see section 7.5). They are applied for thread management, garbage collection, and debugging. The Token Machine also helps the Thread Scheduler to synchronize threads.
- The Token Machine collects part of the root set of reachable objects for garbage collection (see section 7.6).
- Basic debugging features like breakpoints or code stepping are supported.
- Program execution is profiled to identify loops which consume a high amount of execution time. These loops are used as candidates for online CGRA context synthesis.

Most parts of the Token Machine have been implemented newly for DOJA. Only profiling hardware [16, 78] and caches for NAX tables have been retained. Figure 7.5 shows the structure of the Token Machine. It can be divided into two parts roughly: instruction

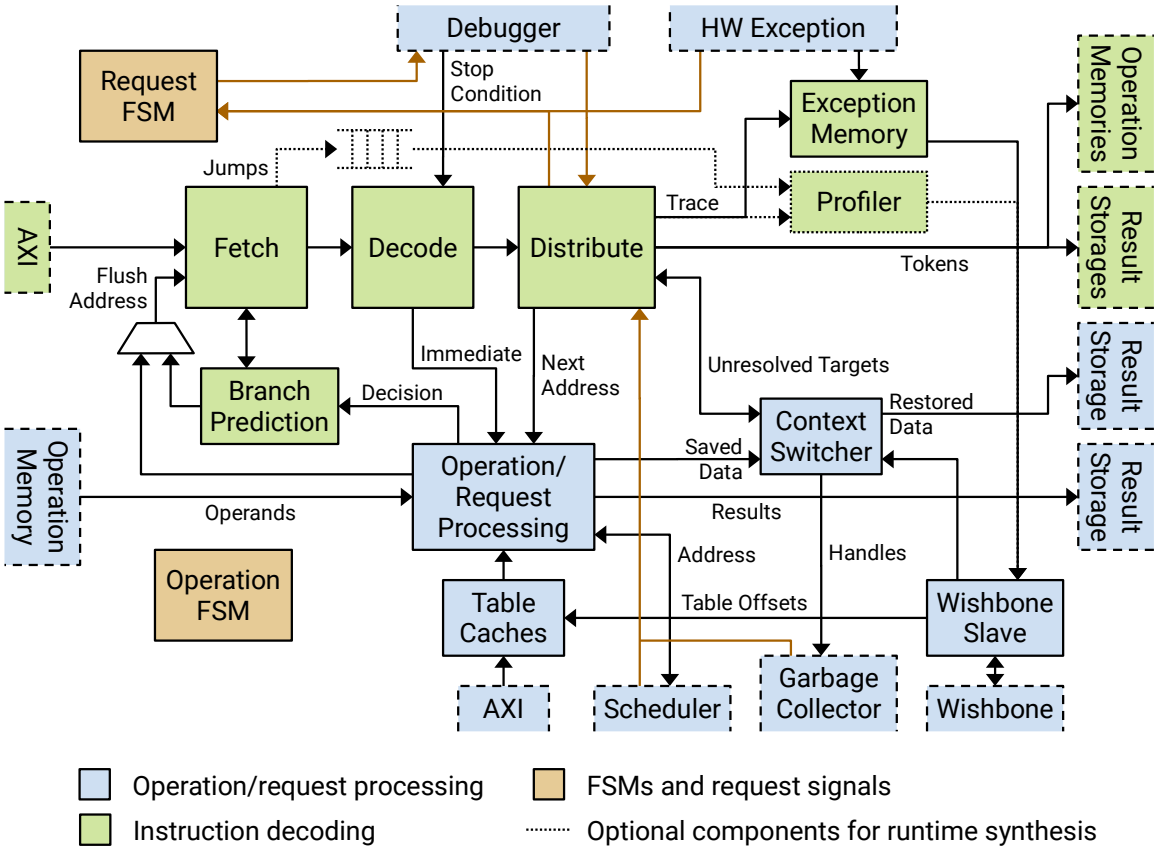


Figure 7.5.: Token Machine overview

decoding (green) and processing of FU operations or external requests (blue). However, it is not possible to draw a clear line between them because they interact closely.

The decoding pipeline will be covered by section 7.2.1. When flushing the pipeline, a start address is given to the fetch stage. Flushing is mainly caused by method invocations, thread switches, and wrongly predicted branches. Instruction fetching is assisted by a simple, stateless branch prediction. Forward branches are predicted not to be taken, backward branches are predicted to be taken. The distribute stage has a trace output, which provides code addresses, FU numbers, and operation addresses of all instructions which have been distributed. This information is stored into the exception memory. If a hardware exception is signaled by an FU, this memory allows to identify the code address which has caused the exception. If runtime CGRA context synthesis is used, the profiler is attached to this interface as well. Jump information (jump offset, conditional/unconditional) is delivered from the fetch stage and is buffered in a queue.

Besides the decoding pipeline, the Token Machine has a data path for processing FU operations and external requests. Operands are taken from the operation memory. Immediate values are delivered by the decode stage. The code address which is waiting to be executed next is provided by the distribute stage. The Thread Scheduler delivers the next code address for thread switches. Access to NAX tables and the constant pool (see section 6.7) is offered by separate caches. They share a common AXI interface. The data path computes code addresses for pipeline flushing or it computes results to be sent to other FUs through result storage. This data path is controlled by two FSMs, which can operate concurrently. One FSM controls operation processing, one FSM controls

request processing. External requests can be received from the Thread Scheduler, the Heap (garbage collector), the Debugger, or the hardware exception interface. Requests are synchronized to the instruction stream by the distribute stage because instructions can also trigger internal events like method invocations or break points. Afterwards, the request FSM handles the requests. Synchronization with the instruction stream is not required in case of hardware exceptions because the provoking instructions have already passed the pipeline.

When switching thread contexts, data waiting in result storages must be saved (see section 7.5). Saving and restoring this data is managed by the context switcher. It stores data of inactive threads in an internal context memory. A separate result storage is used for restoring data during context switching in order to avoid deadlocks. Unresolved target information belonging to this data is received from the distribute stage while saving and is delivered to the distribute stage while restoring. This mechanism is used for finding handles on the data interconnect as well (see section 7.6). The context switcher sends these handles to the garbage collector.

The Wishbone interface provides the offsets of the NAX tables. Furthermore, it allows to manipulate thread context storage. Additionally, profiling and hardware exception information can be read. The profiler has been extended by a programmable code address comparator. Its reference address is written by the CGRA context synthesis thread over Wishbone when code patching is required. The comparator issues an interrupt when control flow leaves the kernel to patch. In consequence, an IST is triggered which patches code memory and invalidates the instruction cache. This hardware extension allows safe and fast code patching.

7.2.1. Decoding Pipeline

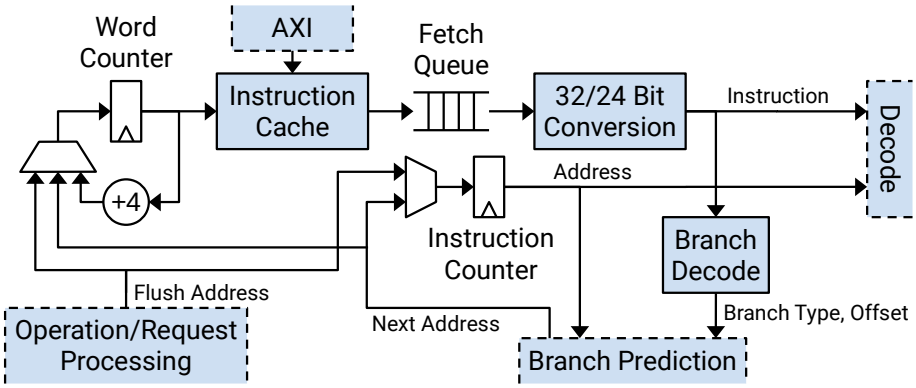


Figure 7.6.: Fetch stage

The first stage of the decoding pipeline fetches instructions as shown in fig. 7.6. The word counter is one of the few components in the processor which run autonomously. Almost all other components are triggered indirectly by this word counter through valid flags. It requests 32 bit words from the instruction cache. The cache is filled via AXI interface and has one clock cycle latency in case of a hit. Cache parameters are given in appendix B. Fetched words are buffered in a queue. The stream of 32 bit words is converted into a stream of 24 bit instructions, which are delivered to the decode stage.

Instruction addresses are generated by the instruction counter. The next value of this counter is defined by branch prediction. If a jump is predicted, the fetch queue is cleared and the word counter is updated. The flush address overwrites both counter values on external pipeline flushes.

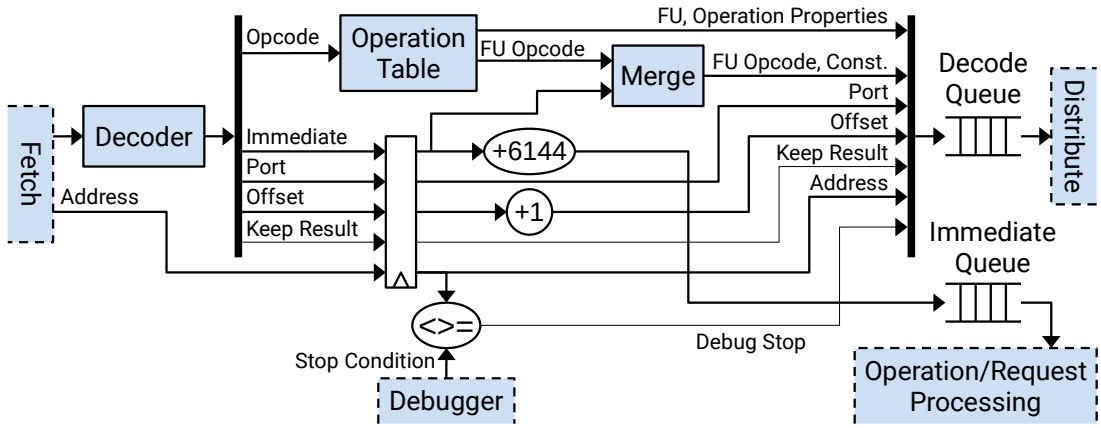


Figure 7.7.: Decode stage

Name	Width	Description
Opcode	7 bit	FU specific operation code
FU	4 bit	Number of the FU to execute the operation
Nop	1 bit	No FU operation needs to be executed.
Has Result	1 bit	The operation produces a result.
Wide Result	1 bit	The result is 64 bit wide.
Inline Param	1 bit	The operation requires a constant operand to be sent along with the operation code.
Hold	1 bit	The distribute stage is halted after the instruction.
Scheduler	1 bit	The distribute stage waits for a scheduler request after the instruction.
Redirect Port	1 bit	All operands are routed to port 0.

Table 7.1.: Structure of an operation table entry

The second stage decodes instructions as illustrated in fig. 7.7. The first step is to separate instruction fields and to generate unique instruction operation codes, which are used as addresses for the operation table. This table contains a fixed mapping from instruction operation codes to FUs and FU operation codes. In addition, it provides several operation properties which are listed in table 7.1. The content of the operation table is generated from configuration files when the processor is instantiated. In principle, the fixed mapping can be replaced by a dynamic mapping if multiple FUs of the same type are available. Since the table is realized by a RAM block, read access has one clock cycle latency. FU operation codes and immediate values are merged for operations which use the *Imm6* field as constant operand. Values contained in `imm` instructions are stored in a queue after 6144 has been added. This queue is read by operation processing. The instruction offset is incremented by 1. The stop condition for debugger stepping is also

checked in this stage. All decoded instruction information is stored into a queue before it is delivered to the distribute stage.

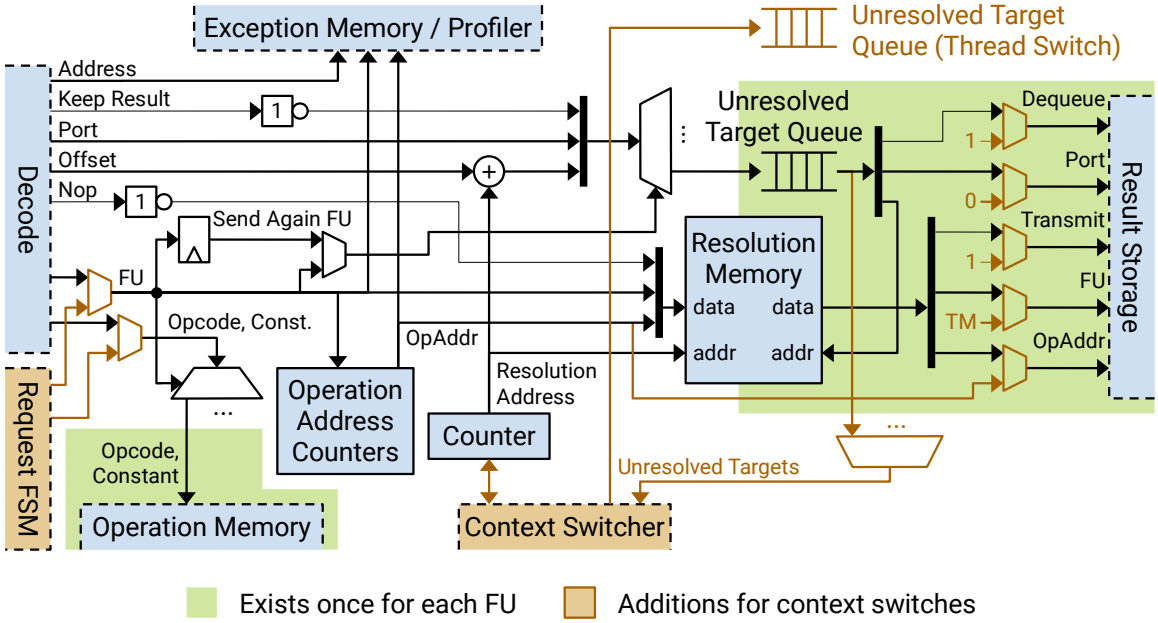


Figure 7.8.: Distribute stage

Token distribution is depicted in fig. 7.8. It happens in two steps for each instruction. First, the operation code is sent to the corresponding operation memory. Second, result target references pointing to this instruction are resolved, which causes target information to be transferred to the result storages of the source FUs. These steps are explained in more detail now. The *keep result* flag is inverted to generate the *dequeue* signal (used for duplicating results). The *nop* flag is inverted to generate the *transmit* signal (used for discarding results, see section 7.1). The operation code and the optional constant are transferred directly to the operation memory of the selected FU. The distribute stage has an operation address counter for each FU. These counters are in sync with the write pointers of operation memories. Therefore, the operation address which must be used for sending operands to the current instruction is known. This operation address is delivered to the exception memory together with instruction address and FU number. At the same time, operation address, FU number, and *transmit* flag are stored into the *resolution memory*. This memory has one global write port and one read port per FU. The write address is called *resolution address* of the instruction. It is generated by a counter. Consequently, the resolution memory is a circular buffer for logging operand matching information of instructions. The resolution address has one more bit than required for addressing. The highest bit is stored as tag of each memory entry. This allows to distinguish between entries of two successive rounds. Adding the instruction offset and the resolution address of the current instruction yields the resolution address of the target instruction. If the current instruction produces a result, this target resolution address is stored into the *unresolved target queue* of the FU together with port and *dequeue* signal. Such queues exist for each FU. The elements at the front of the queues provide the read addresses for the resolution memory. When the target instruction is written into the memory, which can be detected by the tag bit, resolution becomes possible. Then, the resolved entries are removed from unresolved target queues and complete target information is sent to result storages.

An additional register is required for `send_again` instructions because they depend on the last instructions with *keep result* flags (see section 6.1). If this flag is set, the FU number is stored into a special register. The next `send_again` instructions uses this FU number instead of the number received from the decode stage. Since `send_again` instructions generate target information but no FU operations, this modification is necessary for unresolved target queue selection only.

Thread context switches (see section 7.5) require additional hardware. The request FSM must be able to send custom FU operations. Furthermore, results must be redirected to the Token Machine by overwriting target information. Unresolved targets must be shifted from unresolved target queues to the context switcher. They are restored into a special queue which belongs to the separate result storage for context switching. Finally, the resolution address counter must be saved and restored. The FU number which is stored for `send_again` instructions is not saved on context switches because the code generator ensures that the distance between instructions with *keep result* flags and corresponding `send_again` instructions is small (typically less than five instructions). Context switches are delayed after reaching a *keep result* flag and before reaching the `send_again`.

All conditions for stopping the pipeline like method invocations, thread switches, break points, etc. are checked at the entry of the distribute stage. Later checking is not possible because every instruction which passes the distribute stage will be executed. The processor does not support speculative execution. Earlier checking would lead to wasted clock cycles after starting the pipeline again.

7.3. Parameters Relevant for the ISA

The hardware implementation has several parameters which are important for code generation, e.g. for preventing deadlocks (see section 9.5.4) and for limiting instruction offsets (see section 9.5.5). Consequently, these parameters must be considered part of the ISA. Parameter values assumed during code generation may be smaller than those provided by hardware without causing problems. Furthermore, these parameters are relevant for certain hardware functionality such as thread context switching (see section 7.5) and garbage collection (see section 7.6). The impact of the parameters will be evaluated in section 11.5. The terms and symbols introduced here will appear in the rest of this thesis.

FU-Specific Parameters

- The *result capacity* R_{FU} defines how many operations with results can be stored by an FU after receiving all operands and before sending results. It is the sum of the result queue size and of the number of operations which can be executed in parallel due to the pipeline design of an FU.

- The *operation memory size* O_{FU}.

Each unresolved target queue must have a size of at least $R_{FU} + O_{FU} + 1$ in order to eliminate deadlocks because of blocked unresolved target queues. Target queue sizes must be identical to operation memory sizes.

Because the assignment of operations to FUs should not be fixed at compile time, the more abstract definition of *operation categories* is introduced. Two operations belonging to

different operation categories are guaranteed to be executed on different FUs. The reverse implication is not applicable. Two operations belonging to the same operation category may be executed by the same or different FUs. This allows to define FU parameters for abstract categories instead of individual FUs.

Non-FU-Specific Parameters

- The *resolution capacity C* defines the exclusive upper bound for instruction offsets. It is limited by the binary instruction format and the number of addresses in the resolution memory.

7.4. Hardware Exceptions

Each FU can raise hardware exceptions by sending the exception ID and the operation address of the causing operation to the Token Machine over a dedicated interface. FU number and operation address are used for finding the corresponding instruction address which has caused the exception (cause instruction). This mapping is provided by the exception memory inside the Token Machine. At the same time, data interconnect, TDN, and the distribute stage are reset. The decoding pipeline is flushed with the code address of the hardware exception handler. This assembler routine reads the address of the cause instruction via Wishbone interface of the Token Machine. Then, it calls a method written in Java as if it had been invoked by the cause instruction. The Java method obtains FU number and exception ID to throw a suitable exception object, which starts the software exception handling mechanism (see section 6.5).

Although the exact causes of hardware exceptions can be identified, hardware exceptions are not fully precise. At the time when the data interconnect is reset, state changes which follow the cause instruction in the instruction stream might have already been executed. This can result in variables with unexpected values. Hardware exceptions are usually caused by unintentional software defects, which bring the program in an undefined state anyway. Expected failure conditions due to erroneous data input should be checked and handled explicitly instead. Finding software defects is not hindered because the causing instructions can be identified precisely. Consequently, the provided support for hardware exceptions does not constitute a severe limitation in practice.

7.5. Multithreading

Thread context switching with DOJA is more costly than with Java bytecode because a thread context has more components:

- Code position
- Stack of method frames
- Unresolved result targets and corresponding results (at most C)

All three components are visible to the programmer. Consequently, they must be saved explicitly like register file contents on a register-based architecture. Code position is saved by transferring it to the Thread Scheduler over a dedicated interface. The Thread Scheduler

stores it in an internal table together with other thread management information. Instead of saving whole stacks of method frames, Frame Stack memory is partitioned into multiple stacks. Consequently, the active stack can be switched by a simple pointer change. Saving unresolved result targets and corresponding results is more complicated. Unresolved targets are located in unresolved target queues inside the distribute stage of the Token Machine. Waiting results are located in the result queues of the FUs. There might even be data waiting in operation memories. However, if distribution of new operations is stopped and waiting results are saved, all remaining operations will be processed sooner or later. Consequently, it is sufficient to save result queue contents only.

Saving these results could be avoided by switching threads at code positions where no unresolved targets exist. However, experiments have shown that reaching such a position after a thread switch request could take hundreds of clock cycles. Such positions could be inserted explicitly during code generation to shorten waiting time. Because a mechanism for bringing the processor into a “safe” state is required for debugging as well, a generic mechanism for saving and restoring thread contexts has been developed instead.

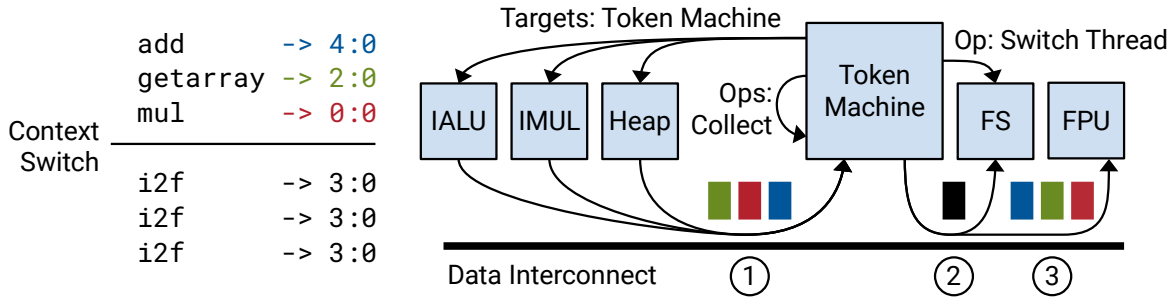


Figure 7.9.: Example for context switch [9]

An example for the context switching process [9] is shown in fig. 7.9. Context switch happens after the `mul` instruction. This leaves three unresolved target references, which target the `i2f` instructions after the context switch. Corresponding results wait in the result queues of Integer ALU (IALU), Integer Multiplier (IMUL), and Heap. ① When a context switch is requested, the Token Machine stops the decoding pipeline and triggers the collection process. This process is implemented in the distribute stage. It sends all unresolved targets to the context switcher module internally (see section 7.2). A `collect` operation is generated for each unresolved target. Corresponding target information is delivered to all FUs with waiting results. This causes the data to be transmitted to the Token Machine where it is stored in the context switcher module as well. ② Afterwards, the Token Machine sends a `threadswitch` operation to the Frame Stack (FS). The corresponding thread ID is transmitted over the data interconnect. This causes the Frame Stack to switch its internal thread context. ③ Finally, the distribution process begins. The context switcher module of the Token Machine loads the stored data into its result queue. The unresolved targets are loaded into the corresponding unresolved target queue. When the decoding pipeline is started again, these references will be resolved automatically. The restored data will be sent to its original target, the Floating Point Unit (FPU) in this example.

Two measures are required to avoid deadlocks. As already stated in section 7.2, data is distributed using a separate result storage and a separate unresolved target queue. Both must have the size C. Furthermore, data must be distributed in the same order as the corresponding result references will be resolved. In the example, the result target of the

multiplication will be resolved first. Consequently, this data word must be distributed first as well. In general, the required order can be different from the order of collection. Therefore, the thread switcher module must reorder data between collection and distribution. This can be done in background during normal execution. Orders are indicated by colors in fig. 7.9.

Context switches can be triggered externally by Thread Scheduler requests due to timeouts. They can also be triggered by special instructions like `monitorenter`. A thread switch is not always necessary with `monitorenter` because the monitor might be free or already owned by the thread. Consequently, a new signal has been introduced from Thread Scheduler to Token Machine. It instructs the Token Machine to skip the context switching process. This is not possible in case of external requests.

7.6. Garbage Collection

The first step for garbage collection is to collect the root set of reachable objects. With Java bytecode, this set is composed of all objects which are referenced by static fields, local variables, and the operand stack. Consequently, the garbage collector has to examine static fields and the Frame Stack memory for handles. With DOJA, the root set comprises handles in operation memories, result queues, and the data interconnect as well. These locations must also be examined for handles.

To solve this problem, handle flags are assigned to all handles which leave the Heap. The operations `getfieldh`, `getfieldh_i` and `getarrayh` have been introduced for this purpose. The flags are carried along with handles in almost all locations where handles can appear outside the Heap: operation memories, result queues, data interconnect, Frame Stack, Forward Unit, and Token Machine. The CGRA does not retain handle flags internally. They are restored for live-out values using the `cgra_pullh` operation.

The context switcher inside the Token Machine stores data of inactive threads including handles (see section 7.5). Therefore, storing data of the active thread to the context switcher temporarily is the easiest method to examine this data for handles as well. Consequently, a new connection from the garbage collector, which is located inside the Heap, to the Token Machine has been introduced for root set collection. If garbage collection requests the root set handles, the Token Machine stops execution and collects remaining data into the context switcher. Then, the context switcher sends all data words which are marked as handles to the garbage collector. Afterwards, data is distributed to the target FUs again without switching threads. Finally, normal execution can continue.

The bytecode-based processor has blocked Heap operation during root set collection. However, the collection of remaining interconnect data by the Token Machine requires all remaining FU operations to be processed. The Heap implementation has been changed accordingly. If garbage collection is triggered, enough free handles (at least C) are guaranteed to handle all remaining allocation requests. In addition, switching memory sections before starting root set collection provides enough free memory. In theory, allocation can fail if the new section does not contain enough contiguous free space. Although the new section is compacted, objects with locked memory addresses can cause fragmentation. Sweeping the old section and switching back is not possible because this depends on root set collection. However, this situation is very unlikely to occur. It has not been a problem with any tested applications so far. All allocation failures lead to hardware exceptions. Handling opportunities are limited because no new objects can be allocated. Nevertheless,

suitable exception objects or strings for printing to standard output can be allocated as a precaution in static initializers.

8. Assembling a Program

Hardware components are only one half of an ISA implementation. The software tool chain is the second half. This chapter gives a short overview of software tool chain components with focus on the assembly language. Chapter 9 will explain conversion from Java bytecode to DOJA code, which will be called *transpilation* from here on.

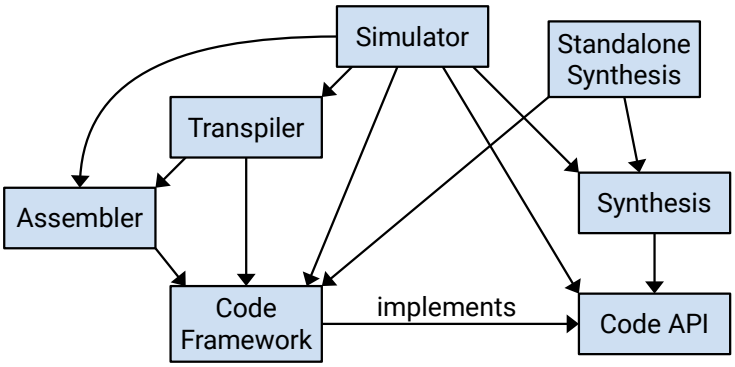


Figure 8.1.: Dependencies between software tool chain components

Figure 8.1 depicts dependencies between software tool chain components. The code API is the basis, which models a complete program as set of Java interfaces. The design of these interfaces is close to the structure of NAX images. The code framework provides implementations for this API. In addition, it contains basic means for code generation and modification. Assembler and transpiler use the code framework as common back end. CGRA context synthesis depends on the code API. This API is also available to runtime software together with a lightweight implementation based on physical memory access. Consequently, synthesis can run on both the developer system and the developed system without considerable modifications. A standalone synthesis tool is available on the developer system, which uses the code framework to synthesize CGRA contexts from loops and to patch them directly into NAX images. An architecture simulator exists, which makes use of the other components to generate programs and to simulate them with or without synthesis. It has not been used for final evaluation because of less precise timings compared to HDL simulations. Consequently, it is not discussed further in this thesis. However, it is a valuable means for verifying other tool chain components.

8.1. Code Framework

The code framework implements the code API to model complete DOJA programs as Java classes. These models can be serialized to NAX files and additional debug information. Conversely, NAX files can be loaded together with corresponding debug information.

Furthermore, the framework is responsible for generating NAX tables. It serves as common back end for all tools which generate or load DOJA programs.

Definition of instruction operations and ISA parameters is also part of the framework. They are configured by a YAML [79] file. This makes modifications or additions to the ISA very convenient. Operations have the following properties:

- Mnemonic
- Operation code
- Used ports and their bit widths
- Bit width of the result
- Instruction format
- Whether the constant operand of R-type instructions is used and whether it is signed
- Whether the operation is stateful
- Whether operands are commutative
- Operation category
- Keywords which help the tools to identify special operations such as invocations and data forwarding

Operation categories have the following properties:

- Name
- Result capacity R
- Operation memory size O

The following generic ISA parameters can be configured:

- Operand matching mechanism
- Resolution Capacity C
- Whether dedicated scratch pad memory separate from LVs is used
- Scratch pad memory size

8.2. Assembly Language

A new assembly language has been specified for DOJA. It helps to test and evaluate ISA features without relying on conversion from Java bytecode. Furthermore, some management functions for exception handling and thread initialization cannot be expressed in Java. These functions can be realized in software using the assembly language instead. Its syntax is based on Jasmin [80, 81], an assembly language for Java bytecode. The parser has been created with JavaCC [82]. Generated syntax trees are transformed directly into the program representation provided by the code framework. Listing 2 contains the important part of the extended Backus-Naur form (EBNF) for DOJA assembly files. Missing nonterminals are defined as follows:

`EOL`	End of line character (new line, line break)
`ID`	Identifier as defined for Java
`PostiveIntLiteral`	Positive integer literal as defined for Java
`IntLiteral`	Integer literal as defined for Java
`FloatLiteral`	Floating point literal as defined for Java
`CharLiteral`	Character literal as defined for Java
`StringLiteral`	String literal as defined for Java

```
File        = { EOL } Class ClassBody ;

ClassBody   = { Field | Method | Super | Implements | EOL } ;

Class       = ".class"
              { "public" | "final" | "abstract" | "interface" }
              Name EOL ;

Super       = ".super" Name EOL ;

Implements  = ".implements" Name EOL ;

Field       = ".field"
              {   "public" | "private" | "protected"
                | "static" | "final" | "volatile" | "transient" }
              ID Type EOL ;

Method      = ".method"
              {   "public" | "private" | "protected"
                | "static" | "final" | "abstract"   }
              MethodId Signature EOL MethodBody ".endmethod" EOL ;

MethodBody  = { Locals | StackArgs | Catch | Statement | EOL } ;

Locals      = ".locals" PositiveIntLiteral EOL ;

StackArgs   = ".stackargs" PositiveIntLiteral EOL ;

Catch       = ".catch" Name "from" Label "to" Label "using" Label ;

Statement   = [ Label ":" { EOL } ] Instruction ;

Instruction = ID [ Label | Immediate ] [ "->" Receiver ] ;

Receiver    = PositiveIntLiteral ":" PositiveIntLiteral [ "+" ] ;

Immediate   =   IntLiteral | FloatLiteral | CharacterLiteral
              | StringLiteral | ( "AMTI" MethodName Signature )
              | ( "RMTI" MethodName Signature )
              | ( "INT" MethodName Signature ) | ( "OFF" Name )
              | ( "CTI" Type ) | ( "SIZE" Name ) | "FRAME_INV"
              | "FRAME_RET" ;

Label       = ID ;

Signature   = "(" { Type } ")" ( "V" | Type ) ;

Type        = { "[" }
              (   "B" | "C" | "D" | "F" | "I" | "J"
                | "S" | "Z" | ("L" Name ";" )        ) ;

MethodName  = { ID "/" } MethodId ;

Name        = { ID "/" } ID ;

MethodId    = ID | ( "<" ID ">" ) ;
```

Listing 2: Important part of the EBNF for DOJA assembly files

Each file specifies one class or interface with fields, methods, super class, and implemented interfaces. Instruction types are not distinguished on grammar level. Instead, context analysis checks whether the instruction components fit the ISA configuration. Special keywords for immediate values allow constants which depend on class structures to be inserted automatically. `AMTI`, `RMTI`, and `INT` insert parameters for static, virtual, and interface method calls respectively. `OFF` obtains a field offset, `CTI` a class table index, and `SIZE` the size of a class. `FRAME_INV` and `FRAME_RET` provide constants for stack frame management. The constant pool is allocated automatically as well. If an `imm` targets a `const` instruction, the value specified as part of the `imm` instruction is inserted into the constant pool and the corresponding pool index is placed into the `imm` instruction. Managing result target references manually is a hard challenge. Therefore, the assembler performs basic checks whether instructions receive all required operands.

The assembler tool flow is illustrated in fig. 8.2. One assembly file per class is required as input. In addition, the ISA configuration must be given (see section 8.1). The resulting NAX image can be loaded into the memory of an AMIDAR system for execution. Additional machine and human readable debug files are created.

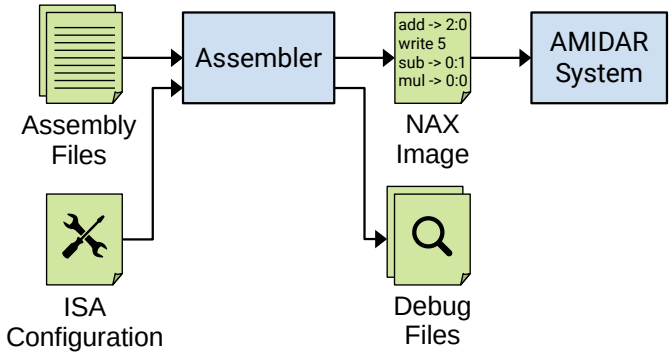


Figure 8.2.: Tool flow for assembling programs for DOJA-based systems

9. Transpilation

Programming with the assembly language is by far too effortful. For running arbitrary Java programs, it is necessary to convert Java class files to NAX images, which implies conversion from Java bytecode to DOJA code. This process will be called *transpilation*. In contrast to compilation, which lowers the abstraction level during conversion, transpilation converts to a representation of same abstraction level. Several tasks must be accomplished:

- Java bytecode instructions must be mapped to DOJA instructions. This is an easy task because DOJA has been designed to be semantically compatible to Java bytecode.
- As not all program constructs are supported natively in hardware, some of them must be replaced by software implementations.
- A valid order of instructions must be found. This is the hardest task because the architecture imposes complex constraints on instruction order.
- Data transfers must be allocated to one of the three opportunities offered by the architecture: direct data transfers, scratch pad memory (SPM), or local variables (LVs).
- Optional optimizations can be performed. Exploring code optimization opportunities is not a research objective. Most implemented optimizations only remove unnecessary code to reduce the size of the final program image.

The transpiler supports separate SPM although it does not exist in the final ISA. Consequently, SPM and LV storage are discussed as separate entities in this chapter. SPM can be compared to register files of traditional architectures. LVs can be compared to stack memory. Hence, traditional allocation algorithms can be applied for those transfer types. Direct data transfers are a new category with special properties. Consequently, a new allocation strategy is required for them. Since direct data transfers are preferred by definition, the transpiler starts with them. Then, problematic direct transfers are shifted to SPM gradually. Afterwards, transfers which do not fit into SPM are shifted to LVs.

The following sections give an overview of the transpilation process. Code transformations are demonstrated with the running example (see listing 1). Instruction scheduling is presented in more detail because it is the most challenging transpilation step.

9.1. Overview

The tool flow for running Java programs on the new DOJA-based AMIDAR system is depicted in fig. 9.1. Java source code is compiled using a standard Java compiler. Resulting class files are processed by a modified version of Retrolambda [20]. This step replaces

dynamic invocations caused by lambda expressions in the source code. Retrolambda applies further transformations in its original version such as replacing default implementations of interface methods. However, these additional transformations are not necessary for transpilation and make code more inefficient. Consequently, they have been removed in the custom version of the tool. Alternatively, Kotlin can be used for programming. If JVM 1.6 is configured as compile target, no further processing of class files is required before transpilation. However, coroutines are currently unsupported because they can result in loops with multiple entry points. In principle, other source code languages which can be compiled to Java class files could be used as well.

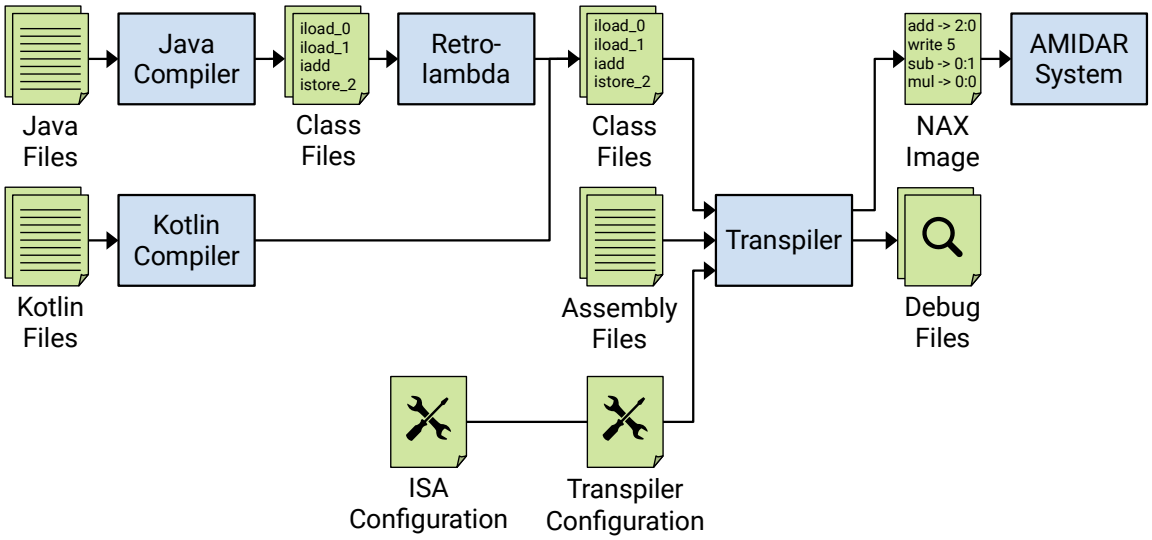


Figure 9.1.: Tool flow for running Java programs on DOJA-based AMIDAR systems

The next step is transpilation. Besides class files, a few assembly files are required as input. They contain code for exception handling and thread initialization. The compiler calls the assembler internally. Classes generated from class files and from assembly files are merged. Additionally, the ISA configuration as described in section 8.1 and a transpiler configuration must be provided. This YAML file contains parameters which normally do not change like the mapping from native methods to DOJA operations. Parameters which are expected to change frequently are passed on command line. Since the code framework is used as back end, transpilation produces the same files as assembling does: the NAX image and complementary debug files.

An overview of the transpilation process is given in fig. 9.2. Details and examples for all steps will be presented in later sections of this chapter. The Java analysis and optimization framework Soot [83, 84] is used as front end. It converts class files to Jimple, which is a representation based on typed 3-address statements. After basic transformations on this representation (see section 9.2), code is converted to Shimple, which extends Jimple by SSA properties. Additional transformations prepare for instruction selection (see section 9.2). Instruction selection replaces each Shimple statement by a set of DOJA instructions (see section 9.3). The resulting instructions are correlated by control and data flow graphs (CDFGs). A CDFG consists of a control flow graph where nodes represent basic blocks. Each basic block contains a directed acyclic graph (DAG) of instructions where edges represent data or state dependencies. These CDFGs are legalized and prepared for scheduling (see section 9.4). Scheduling defines an order of instructions within each basic

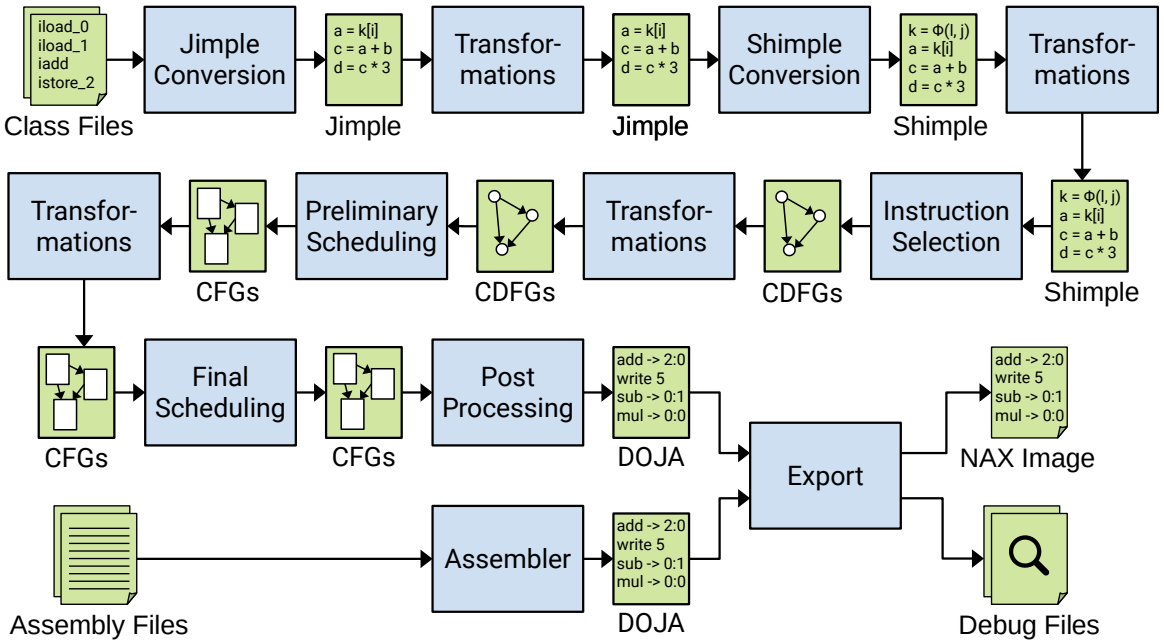


Figure 9.2.: Overview of the transpilation process

block (see section 9.5). Resulting control flow graphs (CFGs) contain a list of instructions for each node. Then, all transformations which require a sequence of instructions can be executed (see section 9.6). A second scheduling step is required because previous transformations might have violated restrictions of the ISA. Finally, CFGs are converted to the program representation of the code framework. Assembly files are converted directly to this representation. The export modules of the code framework produce the final output files.

Figure 9.3 illustrates the main classes which implement the CDFG and CFG representations. “Segment” is synonymous for “basic block” in the following descriptions. Soot models basic blocks differently. Consequently, another name has been chosen to avoid confusion. Methods of a program are represented by `SootMethod` objects. A `SegmentBody` object is assigned to all concrete methods which have an implementation. Such a body contains a `SegmentGraph` which models the CFG. Nodes of the CFG (basic blocks) are realized by `CodeSegment` objects. References between segments define control flow. Normal control flow is distinguished from exceptional control flow by separate references. Each `SegmentBody` holds a list of `SegmentTrap` objects, which specify the exception classes for which exceptional control flow is valid.

The only difference between CDFG and pure CFG representations is how segments store their instructions. In case of a CDFG, instructions are stored in DAGs implemented by the class `Dag`. Each `DagNode` holds one instruction. Nodes have references to their successors and predecessors. They are used to specify data or state dependencies among instructions, which the scheduler must satisfy. After scheduling, instructions are stored in chains instead. `HashChain` is an ordered set of elements provided by Soot.

All instructions implement the interface `AInstr`. Several implementations of this interface exist for different types of instructions. All instructions specify an operation provided by the code framework. Values which are transferred directly between instructions are modeled by `DirectValue`. An instruction has one `DirectUseBox` for each consumed

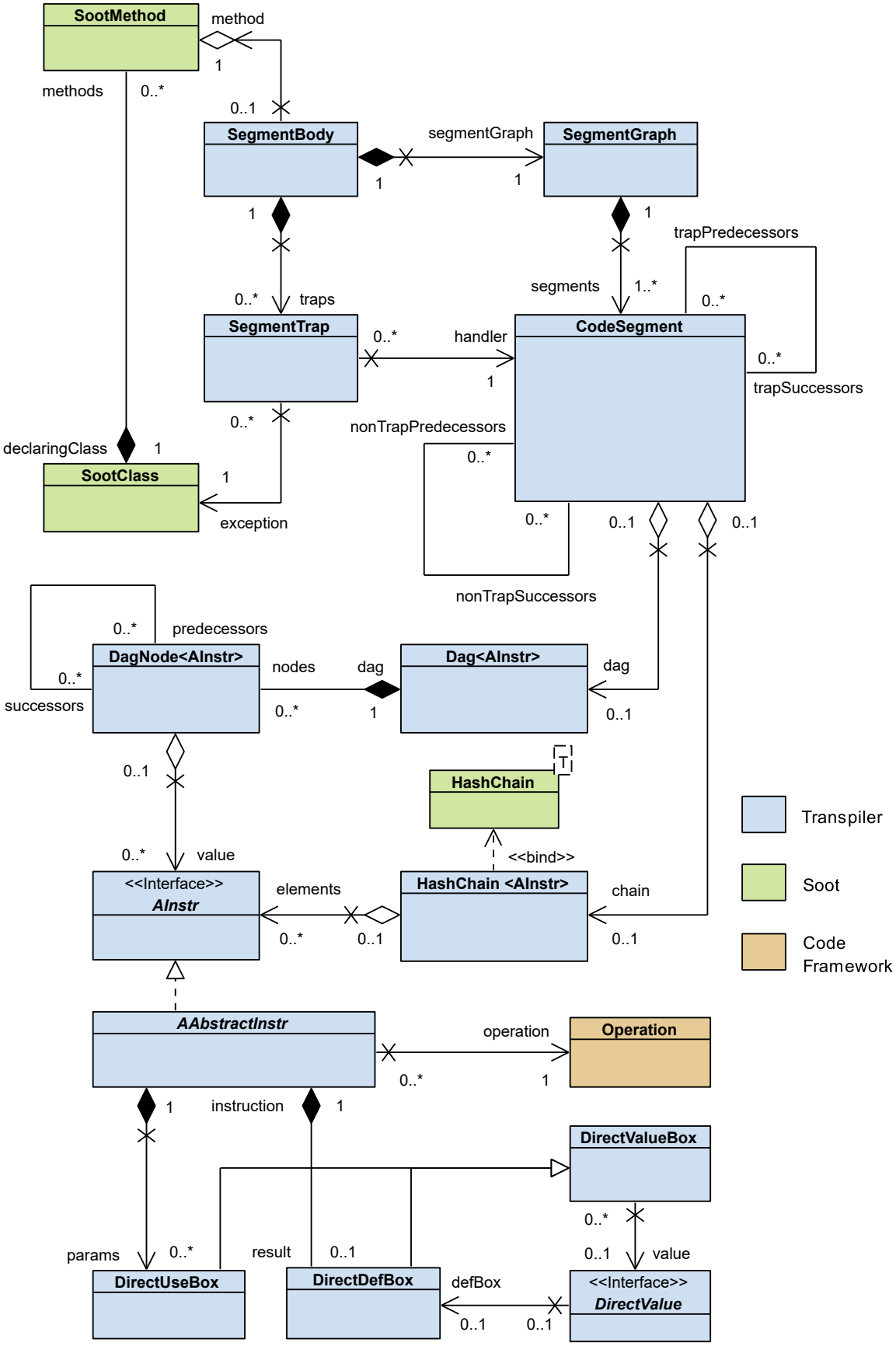


Figure 9.3.: Main classes of CDFG/CFG representations in the transpiler

direct value and one `DirectDefBox` if it produces a result. These value boxes point to the direct values the instruction uses or defines respectively. Since direct data transfers satisfy SSA properties, `DirectDefBox` also provides a reverse reference from value to definition instruction. The additional level of indirection provided by value boxes is inspired by Soot and facilitates code manipulation. Other value and box classes exist for data which is transferred indirectly using LVs or SPM. In general, this intermediate representation is not designed for maximum efficiency in terms of run time or memory consumption. Easy debugging and maintainability is considered more important.

The following descriptions contain the terms Φ function, phi statement, and phi instruction. *Φ function* describes the abstract mathematical concept as known from SSA forms in general. *Phi statements* are corresponding objects of the Shimple representation. A *phi instruction* is a DOJA instruction which consumes different values on one of its ports depending on preceding control flow.

9.2. Jimple and Shimple Transformations

```
1   args := @parameter0: java.lang.String[];
2   a = <app.RunningExample: int[] arrayA>;
3   b = <app.RunningExample: int[] arrayB>;
4   sum = 0;
5   $stack0 = lengthof a;
6   i = $stack0 - 1;
7 label1:
8   if i < 0 goto label2;
9   $stack3 = a[i];
10  $stack2 = b[i];
11  $stack4 = $stack3 * $stack2;
12  sum = sum + $stack4;
13  i = i + -1;
14  goto label1;
15 label2:
16  $stack1 = <java.lang.System: java.io.PrintStream out>;
17  virtualinvoke $stack1.<java.io.PrintStream: void println(int)>(sum);
18  return;
```

Listing 3: Jimple code of the running example after conversion from bytecode

Listing 3 shows the Jimple representation of the running example in text form after conversion from Java bytecode. Jimple is based on statements where each statement reads or writes at most three variables. Method calls are exceptions to this rule because they allow an arbitrary number of method arguments. Variables are typed strictly like in Java source code. Shimple differs to Jimple only in its SSA property. Each variable is assigned once in Shimple code. Phi statements are inserted at points where the value of a variable depends on control flow. Several transformations are performed on these representations.

Remove Unnecessary Assign Statements Statements which assign unused variables and do not have side effects are removed. This happens for optimization levels 1 and above.

Convert Synchronized Methods Methods which are flagged as synchronized are wrapped by synchronized blocks with `monitorenter` and `monitorexit` statements because evaluation of synchronized flags is not supported in hardware.

Replace Switch Statements Switch statements are replaced by binary searches because they are not supported in hardware natively.

Replace Multi-Dimensional Array Allocations Multi-dimensional array allocations are not supported natively either. Therefore, they are replaced by calls to a method which allocates multi-dimensional arrays recursively in software.

Remove Unreachable Methods The call graph analysis of Soot is executed. Methods which can never be reached are removed. Fields and classes which are not used are removed as well. This happens for optimization levels 1 and above.

Devirtualization and Inlining Virtual or interface invocations which have only one call target are replaced by static invocations. Small methods which are called statically are inlined. These optimizations are performed for optimization levels 3 and above.

Remove Unnecessary Jumps Unnecessary jumps are removed in optimization levels 3 and above.

Conversion to Shimple Jimple representations are converted into the SSA variant Shimple.

Constant Propagation Constants are propagated to the statements which use them in optimization levels 1 and above. Constant folding, e.g. replacing the sum of two constants with a new constant, is executed in optimization levels 3 and above.

Remove Unnecessary Phi Statements Conversion to SSA form may produce phi statements which are unnecessary because the resulting values are used only by other phi statements. Such statements are removed.

Move Constants out of Phi Statements Constants which are placed as parameters directly in phi statements are assigned to temporary variables in the preceding basic blocks instead. This step is required to be able to map phi statements to the instruction set.

Remove Unused Argument References Jimple and Shimple method bodies contain one identity statement for each method argument. An identity statement assigns an argument to a variable. This step removes identity statements if the assigned variables are not used. The remaining identity statements are moved from the beginning of the method closer to the positions where the variables are used.

Replace Static Array Initialization Java bytecode as well as Jimple and Shimple initialize arrays using normal code. This step identifies static initialization code for arrays of primitive types. It replaces such code by constants which hold all initial array values. These constants will be transformed into arrays on immortal heap during instruction selection.

The resulting Shimple code of the running example after these transformations is shown in listing 4. Two phi statements for `i` and `sum` have been added. The identity statement for the method argument has been removed. The constant starting value of `sum` has been moved into the phi statement of line 8 first. Then, it has been moved back to the first basic block at line 5. The numbers after "#" in phi statements indicate the preceding statements in control flow. `#0` refers to line 5, `#1` to line 15.

```
1   a = <app.RunningExample: int[] arrayA>;
2   b = <app.RunningExample: int[] arrayB>;
3   $stack0 = lengthof a;
4   i = $stack0 - 1;
5   $const0 = 0;  (0)
6 label1:
7   i_1 = Phi(i #0, i_2 #1);
8   sum_1 = Phi($const0 #0, sum_2 #1);
9   if i_1 < 0 goto label2;
10  $stack3 = a[i_1];
11  $stack2 = b[i_1];
12  $stack4 = $stack3 * $stack2;
13  sum_2 = sum_1 + $stack4;
14  i_2 = i_1 + -1;
15  goto label1;  (1)
16 label2:
17  $stack1 = <java.lang.System: java.io.PrintStream out>;
18  virtualinvoke $stack1.<java.io.PrintStream: void println(int)>(sum_1);
19  return;
```

Listing 4: Shimple code of the running example before instruction selection

9.3. Instruction Selection

Instruction selection converts the Shimple representation of a method to a CDFG of DOJA instructions. The first step is to build a segment graph by extracting control flow. For optimization levels 0 and 1, one segment per source code line number is created, which prevents instruction scheduling across line numbers. Afterwards, the instruction selector iterates over Shimple statements in control flow order and generates instructions for each statement. All variables of the Shimple code are converted to direct values. Instructions are added to the DAGs of their corresponding segments. Edges among instructions of a segment are created to define data and state dependencies. In addition, debug information like variable names and source code line numbers is attached to instructions. Phi statements are replaced by `fwd` instructions. As stated in section 5.1.3, each port of an instruction can behave like a Φ function. Consequently, elimination of phi statements by copying values is normally not required. Exceptions to this rule will be handled later (see section 9.4).

Most Shimple statements can be mapped to single instructions. However, there are some special cases. Method invocations require preparation of method arguments, CTI lookup, and reception of return values. Native methods are replaced by instructions as defined in the transpiler configuration file. Throw statements must be replaced by invocations of the exception handling method. String constants and array constants (statically initialized primitive arrays) are converted into pre-initialized objects on immortal heap. The floating point remainder operation is realized by a combination of multiple instructions.

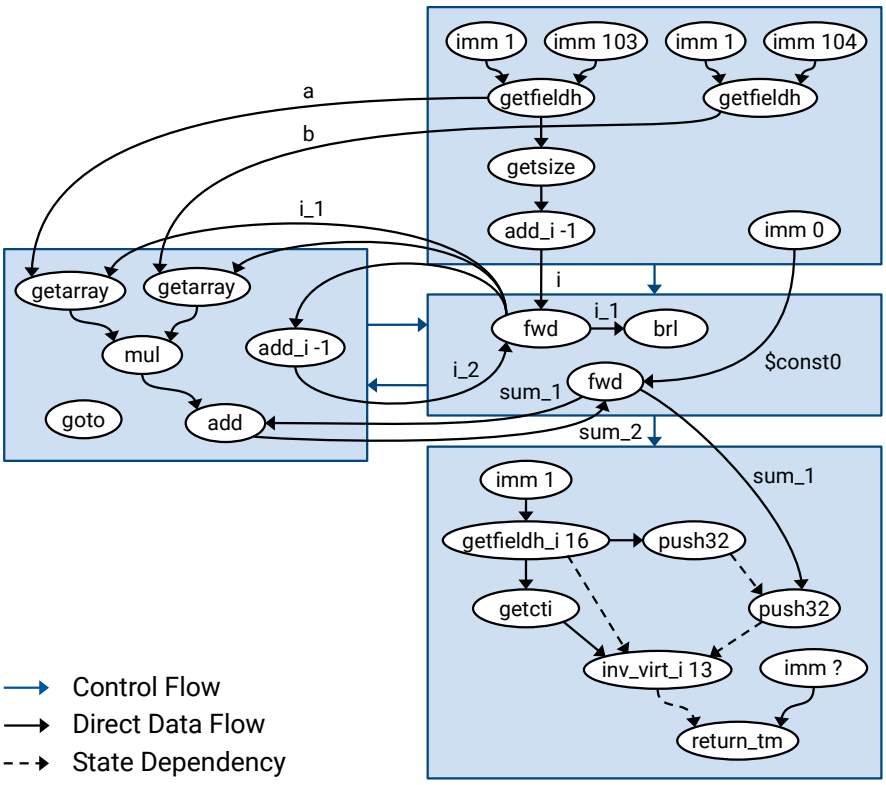


Figure 9.4.: CDFG of the running example after instruction selection

Figure 9.4 illustrates the CDFG which is produced by instruction selection for the running example. Blue elements correspond to the segment graph. Nodes and edges within a segment represent a DAG. Most DAG edges are due to data transfers. State dependencies are required for `push` instructions to enforce the correct order of method arguments. Invocations can imply arbitrary state changes. Therefore, `inv_virt_i` has state dependencies to all stateful instructions of the same segment. Data flow across segments is not modeled by an explicit graph structure. The value objects which are referenced by instructions define data flow instead. `imm ?`, which delivers the operand for `return_tm`, is a placeholder. It will be replaced by stack frame management later.

9.4. CDFG Transformations

After instruction selection, CDFGs are not ready for scheduling yet. Some important dependencies between instructions are missing. Furthermore, values do not comply to the formal characteristics of direct data flow yet (see section 5.1.3). Especially data replication

must be handled. The following transformations are required before scheduling can succeed.

Store Unused Values as LVs With optimization level 0, instructions may exist whose results are not received by any other instruction. However, results must always be consumed. This step stores these results as LVs instead of discarding them in order to simplify software debugging.

Store Exception Handler Live-In Values as LVs Data cannot be passed directly into an exception handler because the software routine for handler search interrupts normal method execution (see section 6.5). Consequently, every value which is live-in to an exception handler must be stored as LV.

Limit the Number of Parallel Direct Values Algorithms with a very high amount of ILP can cause many direct values to be passed between code segments in parallel. However, this amount is limited by the resolution capacity C. Otherwise, it is impossible to specify a target for each of these values. To leave enough room for scheduling, this step limits the amount of parallel direct values to $C - 2$.

Another problem arises if too many phi instructions exist in a segment. Most phi instructions are `fwd` instructions. Too many of these instructions will cause deadlocks which cannot be handled by the scheduler. The amount of phi instructions which can be tolerated is influenced by the result capacity R_{FWD} and the operation memory size O_{FWD} of forwarding operations. Experiments yield $(R_{FWD} + O_{FWD})/2$ as reasonable limit. However, the limit can probably be increased with an improved scheduler.

If one of these limits is reached, direct data transfers are moved to SPM. Those values are chosen which reduce the number of parallel transfers and of phi instructions best. Φ functions are not allowed for scratch pad values. Hence, they must be removed. A modified variant of the algorithm presented by Briggs et al. [85] is used for this purpose. It has been simplified in some aspects because DOJA allows multiple copies to be performed in parallel.

Replicate and Discard Data Values which are consumed more than once on any path of a method are replicated using SPM. Discard instructions are inserted to make sure that all direct values are consumed on every path of a method. Direct values conform to the formal characteristics of direct data transfers (see section 5.1.3) after this step.

Move Long Data Transfers to SPM Values which are transferred over multiple segments are shifted to SPM because the distance will likely exceed the resolution capacity. A limit of 5 segments or 2 branches has been determined experimentally.

Avoid Extended Φ Functions A normal Φ function selects one of its arguments depending on the preceding basic block. However, the value which is consumed by a port of DOJA instructions can depend on the whole preceding program path. Handling such extended Φ functions is complex. It can be avoided by inserting `fwd` instructions purposefully. This is required rarely for practical applications.

Combine Discards The situation can occur that two direct values are received by two discard instructions on one program path and by a single instruction on another program path. The multi-target problem would require the two discard instructions to be scheduled to the same position, which is impossible. Therefore, the discard instructions are combined before scheduling.

Combine Forwards With Target Instructions A forward which has been created for a phi statement of the Shimple code is removed if its result is sent to an instruction of the same segment. The inputs of such a forward are sent directly to the target instruction instead. In principle, this combination could be done across segments as well, which would remove the unnecessary `fwd` in the loop head of the running example. However, this would introduce additional complexity at several points during the transpilation process. Hence, this improvement has been postponed.

Adjust Target Ports This step tackles one half of the multi-target problem (see section 6.3). If direct values are consumed by multiple instructions, they must be consumed on the same port. If an instruction requires an operand only at port 0, this problem can be ignored because data is automatically redirected to port 0 in this case. Ports can be adjusted by leveraging commutativity of operators or by forwarding.

Add State Dependencies for Control Flow Changes Instructions which change control flow must be scheduled at the end of segments. This is achieved by inserting additional state dependencies into DAGs from all leaf nodes to the control flow nodes.

Split Segments After Invocations Segments are split after method invocations. This implies cutting DAGs. The number of direct data transfers on cuts are tried to be minimized. They will be shifted to SPM after scheduling.

The CDFG of the running example after these transformations is depicted in fig. 9.5. Handles `a`, `b`, and `System.out` have been moved to SPM for replication. The value of `i` is replicated by the same technique. `goto` and `brl` have incoming edges which ensure that they are scheduled last. The last segment has been split after the invocation.

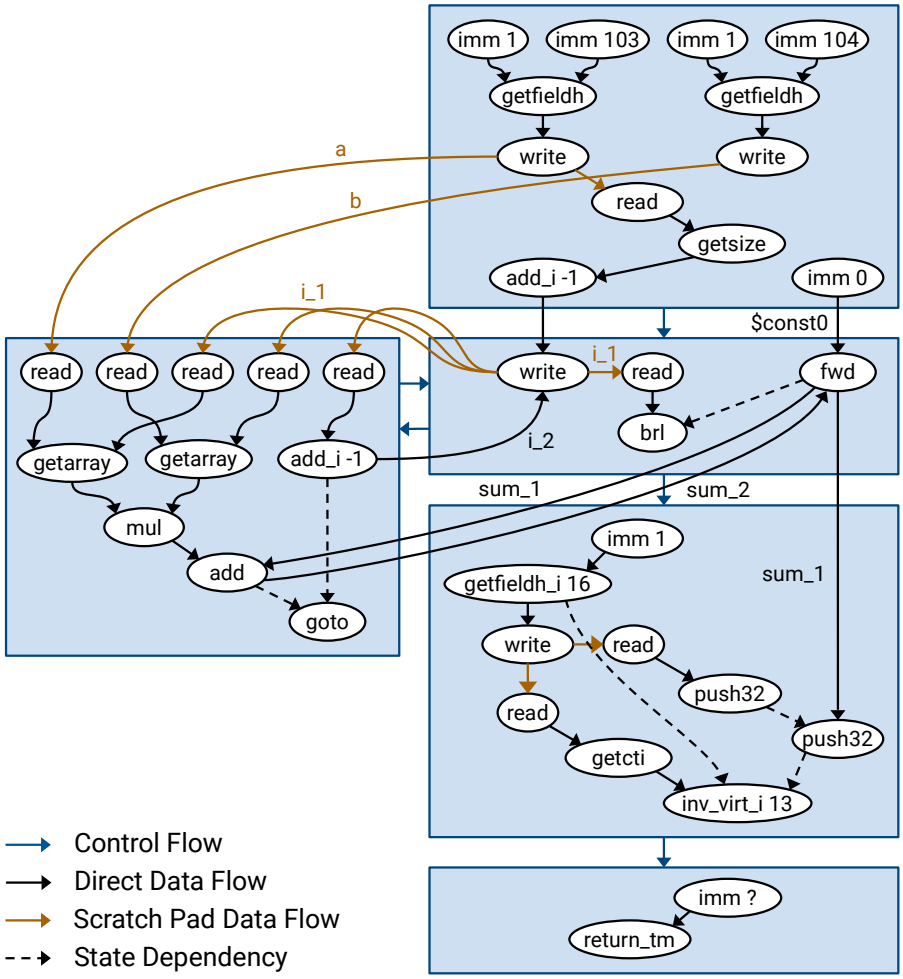


Figure 9.5.: CDFG of the running example before preliminary instruction scheduling

9.5. Instruction Scheduling

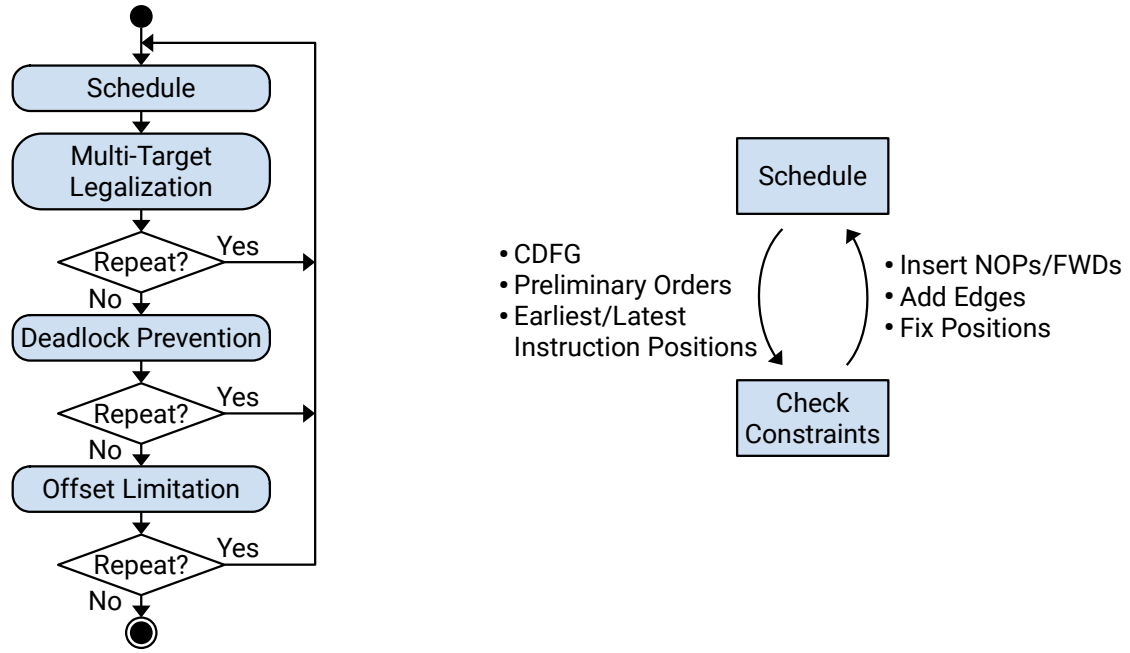
Instruction scheduling takes a CDFG as input. A CDFG consists of a CFG (implemented by a segment graph) where each node (segment) holds a DAG of instructions. The objective is to find topological orders of instructions for all segments. These orders must fulfill additional constraints [8], which make instruction scheduling more challenging in comparison to register-based architectures:

- Instruction offsets must be below the resolution capacity.
- The multi-target problem must be solved. The same instruction offset is required to all targets of an instruction result.
- The code must be free of deadlocks. They would cause the processor to stop operating otherwise.

9.5.1. Global Algorithm

Instruction scheduling can be separated roughly into global scheduling across segments and local scheduling within segments. Both global and local scheduling aim at shortening

instruction offsets. This helps to satisfy the resolution capacity constraint and reduces deadlocks. Figure 9.6a shows the main scheduling loop. It starts with a global scheduling run. Afterwards, constraints are checked. Scheduling as well as constraint checks can request another scheduling repetition by setting the repeat flag. This causes another scheduling run followed by constraint checks to be executed. If all constraints are met, the loop terminates. Figure 9.6b illustrates the interaction between scheduling and constraint checks. The CDFG and the preliminary instruction positions determined by scheduling serve as input for constraint checks. Additionally, estimations for the earliest and latest possible positions of all instructions within each segment are provided. The earliest position is estimated by the first position where the instruction is ready to be scheduled. The latest position is determined by the number of its successors in the DAG and by an additional margin. The margin is influenced by the last scheduling results. Constraint checks can affect the next scheduling run by modifying the CDFG. `nop` and `fwd` instructions can be inserted. Partial instruction orders can be enforced by adding edges to DAGs. Furthermore, instruction positions can be predefined.



(a) Main scheduling loop

(b) Interaction between scheduling and constraint checks

Figure 9.6.: Instruction scheduling overview

Global scheduling is driven by a flow algorithm where the current instruction offsets of pending direct values flow along segments. Pending direct values are direct values which have been sent but have not been received yet. The algorithm starts with local scheduling of the entry segment. The resulting flows are passed to the succeeding segments of the graph as input for local scheduling of these segments. The flow algorithm continues until all segments have been scheduled and flows between segments do not change any more. If a pending direct value arrives at a segment with different offsets due to different paths, the largest and therefore most critical offset is passed on. Multi-target legalization will make sure that offsets are equal during the next scheduling run.

9.5.2. Local Algorithm

List scheduling is used as local algorithm for scheduling individual segments. The algorithm schedules one instruction after the other in forward direction. For each position in the schedule, it determines the instructions which are ready to be scheduled because all their predecessors in the DAG have already been scheduled. If one of these instructions has been fixed at the current position by a constraint check, it is selected immediately. Otherwise, the priorities of all ready instructions are computed. The instruction with the highest priority is selected. A priority is composed of the following components ordered by descending importance.

1. *Special*: If the instruction has missed its fixed position, has an operand whose offset exceeds the resolution capacity, or has an operand with alternative target, this component is set to 1. If the instruction is fixed at a later position or forwards a value for avoiding the resolution capacity limit, this component is set to -1. Otherwise, it is 0.

2. *Urgency*: The urgency value of an instruction helps to keep instruction offsets short. It is incremented for all predecessors of a target of a pending direct value. Figure 9.7 depicts an exemplary DAG. When node 1 is scheduled, the urgencies of nodes 2, 3, and 4 are incremented by 1. Consequently, these nodes will be scheduled before node 6 to allow node 5 to be scheduled as soon as possible.

3. *Direct values*: This component is the sum of the squared offsets of the operands. If the instruction result is live-out from the segment, the squared number of unscheduled nodes in the DAG is subtracted. This computation also aims at keeping instruction offsets small.

4. *ALAP (as late as possible) position*: The negated ALAP position is used for this component. It causes instructions with few successors in the DAG to be scheduled late. For nop instructions, this component is 0 to have them scheduled early because they contribute to solving the multi-target problem.

5. *Previous instruction offset*: This component is computed by negating the target instruction offset of the last scheduling run. If the instruction does not produce a result, the component is 0. Again, this keeps instruction offsets short.

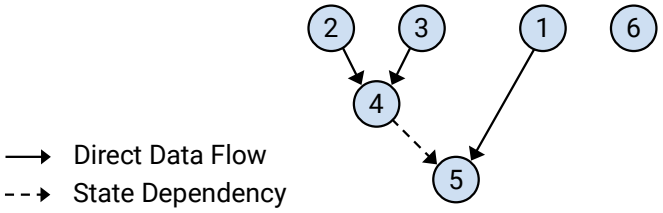
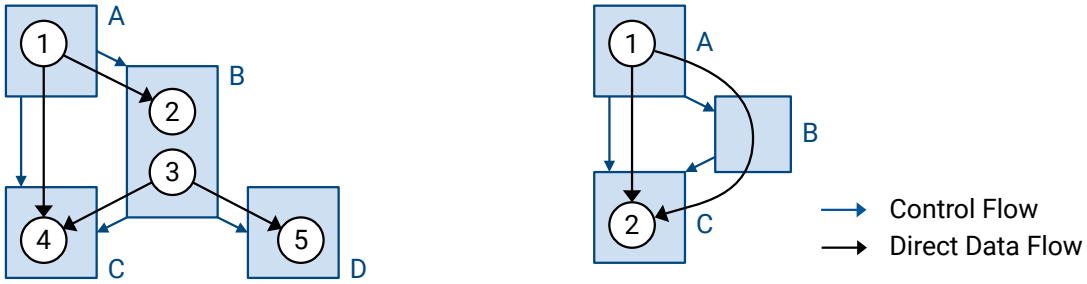


Figure 9.7.: Exemplary DAG for urgency metric

Components are compared individually without calculating a combined metric. Only if the components of highest importance are equal, the components of next lower importance are considered. This priority composition has been found experimentally by analyzing problems which arise during scheduling. After scheduling an instruction, its estimations for earliest and latest positions are adjusted. Additionally, flows and urgencies are updated.

9.5.3. Multi-Target Legalization

The multi-target problem has been introduced in section 6.3. Figure 9.8a shows an abstract example. Instruction 1 sends its result to instructions 2 or 4 depending on the branch decision at the end of segment A. Both instructions must be scheduled at the same position relative to the beginnings of segments B and C. However, the situation is more complicated. If control flows through segment B, instruction 3 sends its result to instructions 4 or 5 depending on the branch decision at the end of segment B. Consequently, instructions 4 and 5 must also be scheduled at the same position relative to the beginnings of sections C and D. Therefore, all three instructions 1, 4, and 5 must be scheduled together.



(a) Related values

(b) Alternative paths to same target

Figure 9.8.: Abstract examples for the multi-target problem

The multi-target legalization identifies all cases where a direct value is sent to multiple alternative targets. It checks whether all targets can be scheduled at suitable positions, which are limited by the estimations for earliest and latest positions of the last scheduling run. If no such position exists, a `fwd` is inserted before the instruction which has the highest distance to the source instruction. If multiple suitable positions exist, the position is chosen which differs from the previous schedule least of all. The target instructions are fixed at the chosen positions, which will be respected by the scheduler during the next run.

A situation as illustrated by fig. 9.8b can appear as well. Although instruction 1 sends its result only to one target, this target can be reached over different paths with different instruction offsets. A `fwd` instruction is inserted into segment B to solve this problem.

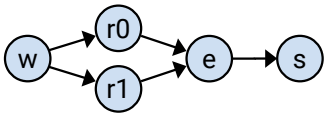
9.5.4. Deadlock Prevention

The main challenge for instruction scheduling is to produce code which is free of deadlocks. In contrast to register-based architectures, certain instruction sequences can cause the processor to stop operating. The first step to a solution is to identify all deadlocks, which can be achieved by static code analysis. The second step is to resolve them by changing instruction orders or by inserting new instructions.

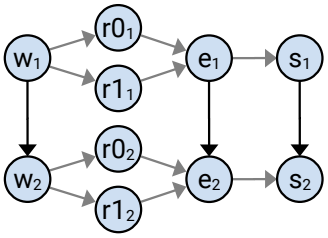
Deadlock Analysis Graphs

Deadlocks are identified by building deadlock analysis graphs from the instruction sequences of the last scheduling run. The nodes of these graphs model execution states of instructions. Edges model dependencies between states. FU-specific ISA parameters (see

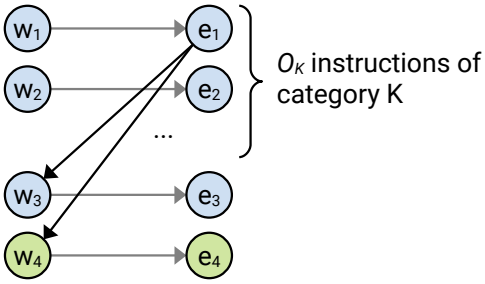
section 7.3) play an important role here. Figure 9.9a depicts a graph for an instruction which receives two operands and sends a result. The *waiting* node (w) represents the state when the instruction has been decoded and stored into an operation memory. *Received 0* (r0) means that the instruction has received the operand for port 0. Likewise, *received 1* (r1) indicates that the instruction has received the operand for port 1. These states can be entered in any order after the instruction has been waiting. They can be active simultaneously. *Executed* (e) means the instruction has been executed on an FU. Execution is possible only after all operands have been received. The *executed* state depends directly on the *waiting* state for instructions which do not require any operands. *Sent* (s) indicates that the instruction has sent its result, which can happen only after the instruction has been executed. This state does not exist for instructions without result.



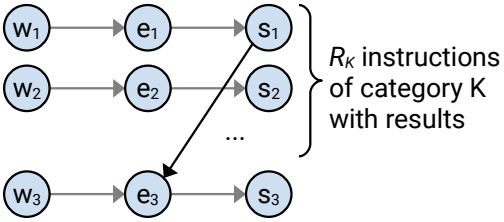
(a) Instruction



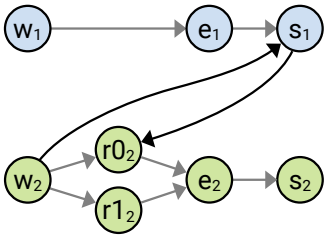
(b) Two instructions of identical category



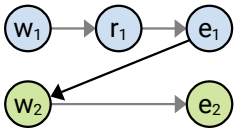
(c) Limited operation memory size



(d) Limited result capacity



(e) Data transfer



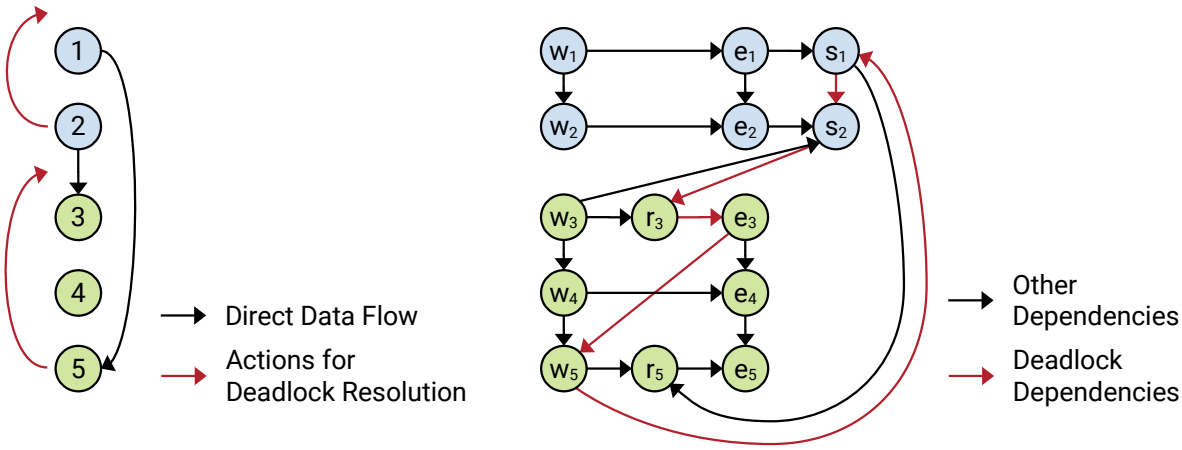
(f) Branch or blocking scheduler operation

Figure 9.9.: Deadlock analysis graphs (colors indicate operation categories)

Dependencies between two instructions with identical operation category are shown in fig. 9.9b. Decoding, execution, and result sending happen in order within a category. Consequently, the corresponding instruction states depend on each other. Operands can be received in arbitrary order.
Limited operation memory sizes O_K introduce additional dependencies as illustrated in fig. 9.9c. Only O_K instructions of operation category K can be waiting in their operation

memory simultaneously. Therefore, a dependency is required from the *executed* node of instruction 1 to the *waiting* node of instruction 3, which follows after O_K instructions of category K. The instructions of category K do not need to follow directly after another. The first instructions of other categories which follow after instruction 3 require similar dependencies because the decoding pipeline is blocked if an operation memory is full.
Limited result capacity R_K must be modeled as depicted in fig. 9.9d. At most R_K results of instructions with category K can be stored in execution pipeline and result queue. Consequently, a dependency is required from the *sent* node of instruction 1 to the *executed* node of instruction 3, which follows after R_K instructions of category K with results.
Each direct data transfer introduces the two dependencies shown in fig. 9.9e. Instruction 1 sends data and instruction 2 receives it at port 0. The *sent* state of instruction 1 can be reached only if instruction 2 is in its *waiting* state. Instruction 2 can enter the *received* state for port 0 only if instruction 1 has reached its *sent* state. Note that the dependency from *waiting* to *sent* state is the only type of edge which is directed "upwards" against the sequence of instructions.
The last type of dependency is caused by branches or blocking scheduler operations such as `monitorenter`. Figure 9.9f illustrates an abstract example. All instructions which follow the branch can become waiting only after the branch has been executed. Hence, dependencies from the *executed* node of branch instruction 1 are required to the first instructions of each category after the branch.

All cycles in deadlock analysis graphs will cause deadlocks. Consequently, deadlocks can be identified by cycle search. An abstract example is given in fig. 9.10. It shows a sequence of instructions and the resulting deadlock analysis graph, which contains a cycle. Before looking at strategies for resolving deadlocks, the difference between operation categories and FUs must be discussed with respect to deadlock detection. Instructions which belong to separate operation categories are guaranteed to be executed on separate FUs. Instructions which belong to the same operation category can be executed on the same FU but they may be executed on separate FUs as well. Therefore, using operation categories instead of precise FU mappings can only introduce new dependencies but cannot reduce them. This ensures that all deadlocks are found.



(a) Instruction sequence (b) Deadlock analysis graph

Figure 9.10.: Deadlock example for $O_K = 2$ (colors indicate operation categories)

Deadlock Resolution

The current deadlock resolution strategy handles each deadlock individually. Whenever a cycle is found in a deadlock analysis graph, the cycle is examined to choose a suitable resolution action. Two types of actions are utilized: reordering instructions by adding edges to segment DAGs or inserting `fwd` instructions. If a w → s → s → r sequence is detected in a deadlock cycle, the order of the instructions belonging to the s nodes is changed. Figure 9.10 shows such a case. Instruction 2 can be placed before instruction 1 to remove the deadlock. If an s → e sequence is part of the deadlock cycle, the instruction belonging to the e node can be placed before the instruction belonging to the s node. In case of an e → w sequence, the w instruction can be positioned before the e instruction. This rule can be applied to the example of fig. 9.10 as well. It would result in instruction 5 to be moved in front of instruction 3. However, changing the instruction order can sometimes be impossible because it is already defined by the CDFG. In this case, inserting a `fwd` instruction often helps. Starting from the w node of the last instruction, the first w → s sequence on the deadlock cycle defines the direct value to be forwarded. This strategy could also be applied to the example of fig. 9.10. The data transfer between instructions 1 and 5 could be directed through a `fwd` instruction. Special handling is required for deadlocks which are caused by direct data flow among too many instructions of same category. They can be resolved by breaking these long dependency chains within a category with the help of `fwd` instructions.

Although, these resolution strategies are applied successfully for all benchmarks, they do not guarantee to remove all deadlocks. Situations exist in which a deadlock is detected but cannot be resolved (see section 11.5). The core problem is that each deadlock is treated individually. Removing one deadlock can create a new one or can make resolution of other deadlocks more difficult. More “global” handling schemes which take multiple deadlocks into account would be beneficial. Further research is required on this topic.

Coping with Complexity

This paragraph has been published previously in [8]. Theoretically, a dependency graph must be constructed for each possible execution path in a program. Calling convention ensures that no deadlocks can appear across method boundaries. Consequently, methods can be analyzed separately. Loops still produce an infinite number of paths. However, result target references are limited to the current or to the next loop iteration. Therefore, no additional deadlocks can appear after analyzing two loop iterations. The number of paths can still grow exponentially. In practice, this problem is solved using a sliding window algorithm. The window slides along the control flow of the method. Whenever the next instruction is added to the window, cycles are searched and removed. Afterwards, instructions which can be proven not to cause new deadlocks are removed from the window. When the algorithm detects that a window position has already been encountered, analysis of this path can be finished. While the problem still has exponential complexity, this algorithm finds all deadlocks in reasonable time even in methods with very complex control flow.

Instructions can be proven not to cause new deadlocks as follows. w → s edges generated by direct data transfers are the only edges which are directed from newer to older instructions. When adding further instructions to the graph, new cycles can only be introduced by new w → s edges. These edges can be caused only by instructions which receive

direct values. Each direct value can be received at most once within a deadlock analysis graph. Therefore, every new cycle will contain an s node of an instruction whose result has not been received yet. All nodes which are not reachable from such s nodes can never be part of a new cycle. They can be removed together with corresponding instructions. Furthermore, the fact that each cycle contains at least one w → s edge simplifies cycle search and deadlock resolution.

9.5.5. Offset Limitation

The offset limitation searches for instruction offsets which exceed the resolution capacity. `fwd` instructions are inserted in all segments where this happens. These `fwd` instructions are tagged to be treated specially during scheduling. They will be scheduled just before the offsets of their operands will reach the resolution capacity.

The final CFG which results from executing the global scheduling algorithm is illustrated in fig. 9.11. All constraints have been satisfied without adding instructions.

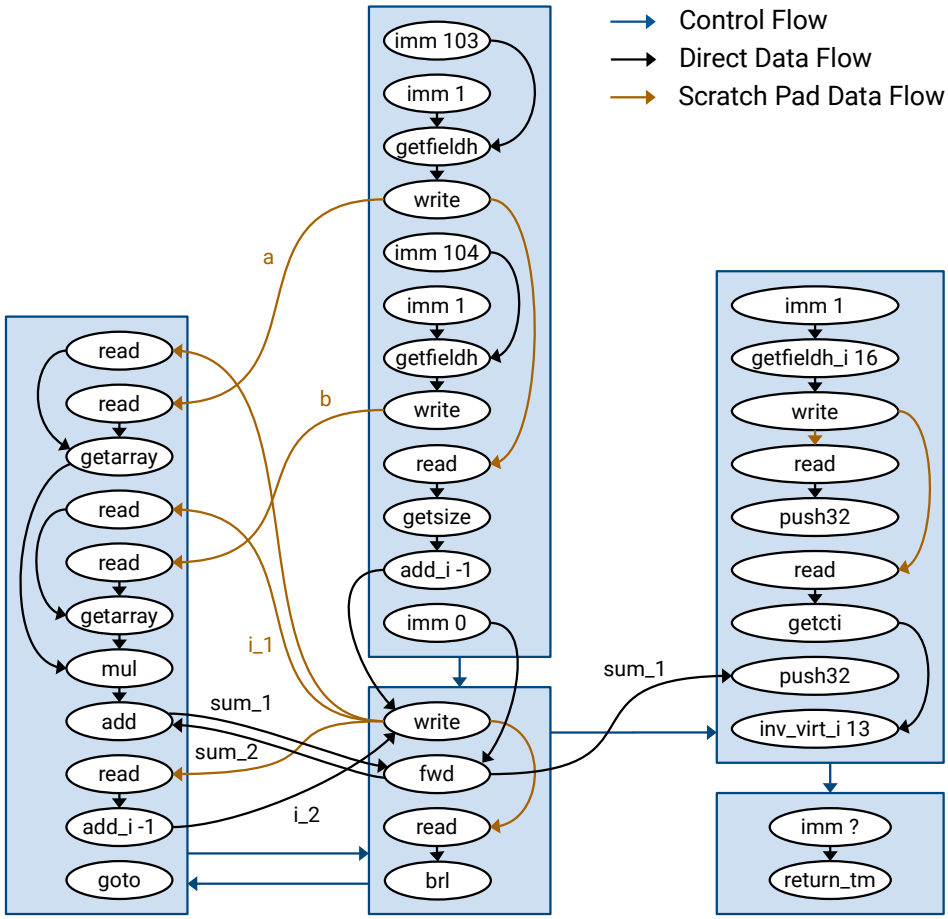


Figure 9.11.: CFG of the running example after preliminary instruction scheduling

9.6. Transformations after Scheduling

After preliminary instruction scheduling, all transformations which require a sequence of instructions can be executed. Direct values which are forwarded many times or which are transferred across method invocations are moved to SPM. Then, values are allocated to SPM and LV slots. Stack management instructions are added. Finally, a second scheduling pass is required because previous transformations might have violated scheduling constraints. The following transformations are performed.

Move Long Forward Chains to SPM Values which are forwarded more than twice are stored in SPM instead. This step aims at reducing data transfers between FUs because each `fwd` introduces one additional data transfer. Using SPM, only one additional data transfer is required independent of the distance between instructions.

Move Values Which Pass by Methods to SPM Direct data transfers cannot pass by method invocations because resolution of result target references continues in the invoked method, not after the invocation. Such values are stored in SPM instead.

Propagate SPM Copies Previous transformations can result in copying SPM contents from one address to another. This step tries to remove such copies. Scratch pad values are replaced by their originals whenever possible.

Replace SPM Values by Direct Data Duplication Up to this step, data is always duplicated using SPM. The alternative mechanism based on keep result flags and `send_again` instructions is applied at this point. Currently, this mechanism is used only within segments. The algorithm searches for scratch pad values which are written once and read twice. If `write` and `read` instructions are positioned close to each other and the constraints mentioned in section 6.1 are met, these SPM instructions are replaced by direct data duplication. This step is executed for optimization levels 2 and above.

Allocate SPM SPM provides only a limited amount of addresses. Hence, scratch pad values must be allocated to these addresses. Values which do not fit into SPM must be stored as LVs instead. The final processor design does not contain a dedicated SPM. `read` and `write` instructions access the same LV slots as `load` and `store` instructions. They differ to `load` and `store` instructions only by specifying LV addresses directly inside instructions instead of utilizing separate instructions for address specification. Consequently, they still provide performance benefits.

Because SPM has similar properties as register files, a traditional algorithm for register allocation can be used. A simple greedy algorithm has been implemented which is inspired by the register allocator of LLVM [86]. However, it supports neither live range splitting nor register coalescing. The algorithm allocates the largest live ranges first. Values with higher spill weight displace values with lower spill weight. Spill weight is the number of uses and definitions divided by the size of the live range.

Assign LVs LV slots are not limited but stack values must be assigned to LV slots efficiently in order to save stack memory. A simple heuristic based on graph coloring is applied. Values conflicted with many other values are allocated first.

Combine Instructions Some pairs of instructions which occur frequently and are executed on the same FUs have own operation codes. They are replaced by equivalent, single instructions. This reduces code size and speeds up execution (see section 11.6). `write` and `read` are replaced by `writer`, `imm` and `load/store` are replaced by `read/write`, `read` and `push32` are replaced by `readpush32`. The later two replacements are only possible without dedicated SPM. All replacements are performed for optimization levels 1 and above.

Add Prologue and Epilogue Prologues and epilogues, which create and discard stack frames, are added. In case of dedicated SPM, they also save and restore SPM contents.

Adjust Target Ports Target ports are adjusted to solve the multi-target problem. This step uses the same techniques as the corresponding CDFG transformation (see section 9.4).

Fix Large Branches Branch offsets which exceed the limit imposed by the binary encoding of B-type instructions (see section 6.6) are stored in `goto` instructions instead.

Final Scheduling A second scheduling pass must be executed because previous transformations might have violated scheduling constraints such as the resolution capacity. Segment DAGs are rebuilt from instruction sequences. Since the assignment of SPM and LV addresses must remain valid, these DAGs contain more dependencies than for the preliminary scheduling pass. They leave less freedom for scheduling.

Append Unwind Handler Unwind handlers (see section 6.5) are appended to each method. They clean up the stack for exception handling.

The final code for the running example is given by fig. 5.2. The stack management instructions `invoke_fs_i` and `return_fs_i` have been added. LV addresses have been inserted into `read` and `write` instructions. `write` and `read` in the loop head have been replaced by `writer`. `write` and `read` before the method invocation have been replaced by direct data duplication with `send_again`. A `fwd` has been added to the loop body during the final scheduling pass.

Part IV.

Evaluation

10. Prerequisites

10.1. Benchmark Applications

Two benchmark sets are used to evaluate the performance of the DOJA processor. This section gives an overview of the contained benchmark applications and their properties. All benchmarks are transpiled with optimization level 2, which enables elimination of unreachable code and instruction scheduling across source code line numbers. No high level optimizations with significant performance impact like method inlining, redundancy elimination, or loop unrolling are applied.

10.1.1. Micro Benchmarks

The set of micro benchmarks comprises 27 small algorithms with very flat call graphs. Their execution times lie in the range of milliseconds for the evaluated systems. Most of them perform arithmetic or logic computations with few control flow jumps and object allocations. These benchmarks are suitable for fast evaluation with HDL simulations in early design stages.

Encryption and Hashing Nine encryption algorithms (AES, Blowfish, DES, IDEA, RC6, Serpent, Skipjack, Twofish, and XTEA) and ten hash algorithms (BLAKE256, CubeHash512, ECOH256, MD5, RadioGatun32, RIPEMD160, SHA1, SHA256, SIMD512, and Whirlpool) are contained in the set. They mainly perform logic 32 bit operations and consist of regular loops.

Image Filters Four image filters (Contrast, Grayscale, Sobel, and Swizzle) are available. They mostly perform arithmetic 32 bit operations and consist of regular loops. Contrast filter uses floating point numbers while the other filters use integer numbers.

Codecs Audio encoding and decoding with ADPCM are contained as separate benchmarks. Primarily, arithmetic 32 bit operations on integer numbers are performed. ADPCM encoding uses some 64 bit integers as well. The algorithms consist of several nested loops. Additional branches handle corner cases and block boundaries.

Image encoding with JPEG is composed of several smaller algorithms like discrete cosine transformation, color-space transformation, and Huffman encoding. Mainly, arithmetic 32 bit floating point operations are applied. Huffman encoding exhibits more complex control flow than the previous algorithms. Additionally, the application is spread across multiple methods.

Regular Expression Matching A simple benchmark for regular expression matching is included as well. This benchmark differs significantly from the previous ones because it is primarily based on control flow jumps and virtual method invocations instead of arithmetic/logic computations.

10.1.2. SPEC JVM98

The SPEC JVM98 benchmark set [39] provides practical and complex work loads. It is composed of eight benchmark applications, which run for minutes on the evaluated systems. In general, they have much more complex control flow and more method invocations in comparison to micro benchmarks. Furthermore, they require a higher amount of input and reference data, which must be provided by a file system. These files are read from an SDHC card which is connected to the processor via SPI. The benchmark set can be executed with three different problem sizes. The largest size of 100 is used for evaluation in this work.

Compress This benchmark compresses files using modified Lempel-Ziv method. It is based on a mixture of logic 32 bit operations, additions, table lookups, and branches. Few objects are allocated and the number of method invocations is lower compared to other SPEC JVM98 benchmarks.

Jess JESS is an expert shell system based on NASA's CLIPS. It makes extensive use of method invocations and branches. Furthermore, it allocates many objects.

Raytrace A single-threaded ray tracing algorithm renders a scene showing a dinosaur. Much 32 bit floating point arithmetic is performed. Each ray is modeled as object with getters and setters, which results in many object allocations and method invocations. This benchmark is not included in the official list but allows evaluation of multithreading together with the *mtrt* benchmark.

DB This benchmark loads a database file and executes operations on the records. Control flow is more important than computations. Memory usage is moderate.

Javac The Java compiler of JDK 1.0.2 compiles a lexical analyzer. This benchmark has a very complex control flow including many virtual method invocations and exception handling. The amount of code is relatively large with 177 classes.

Mpegaudio This application decodes a 3.1 MiB MP3 file. Much 32 bit floating point arithmetic is applied. The number of allocated objects is low compared to other SPEC JVM98 benchmarks.

Mtrt The same ray tracer as in the *raytrace* benchmark is executed but with two parallel render threads. This is the only benchmark which makes significant use of multithreading. Other benchmarks apply multithreading only in form of interrupt service threads.

Jack Jack is an early version of the JavaCC parser generator. It uses the grammar for Jack itself as input. Control flow is very complex and includes exception handling. Furthermore, strings are copied frequently.

10.2. Target Hardware Platform

The target hardware platform in this work is a Digilent Nexys Video evaluation board [87]. It combines a Xilinx Artix 7 FPGA (XC7A200T-1SBG484C) with 512 MiB Micron DDR3 RAM (MT41K256M16HA-187E). FPGA configurations are generated by Vivado 2019.1 with synthesis strategy *Flow_PerfOptimized_high* and implementation strategy *Performance_Retiming*. The desired target frequency of the processing core is 100 MHz.

11. Variants and Parameters

Hardware implementation of a DOJA processor leaves room for several parameters in binary encoding, memory structure, data interconnect, and in the ISA itself. This chapter evaluates the influence of these parameters on performance and hardware resource consumption. Subsequently, parameters are fixed based on the results.

Performing all measurements sequentially in silicon would be very time consuming. Instead, performance evaluation for variants and parameters is accomplished by measuring clock cycles in behavioral HDL simulations, which can be run in parallel. Simulation of SPEC JVM98 benchmarks would take weeks even with the lowest problem size. Consequently, simulated performance is evaluated only for micro benchmarks. Hardware resource requirements are determined by synthesizing FPGA configurations from the designs, which can be parallelized as well. Table 11.1 gives the starting values of the optimization process. They have been found as reasonable values during development. The maximum number of live threads, the maximum number of acquired monitors, and stack size do not influence performance. They are not relevant for optimization but can be chosen according to application requirements. Further parameters which have been kept constant for this work are listed in appendix B.

Parameter	**Value**
Dedicated SPM Addresses	8
Number of Data Buses	6
Operation Memory Sizes	8
Result Queue Sizes	4
Resolution Capacity	16
Max. Number of Live Threads	64
Max. Number of Acquired Monitors	32
Stack Addresses per Thread	2048

Table 11.1.: Starting values of hardware parameters

11.1. Instruction Format

The first parameters which must be defined are the bit widths of instruction fields (see section 6.6). First estimations give 24 bits as the smallest multiple of one byte which can store all relevant information. Two bits are required for encoding the instruction type. One bit is reserved for future extensions. For example, vector instructions could be introduced as separate instruction class. This leaves 21 bits to be distributed to the fields. No FU has more than three ports. Consequently, 2 bits are sufficient for port fields.

116 operations exist for R-type instructions, which leads to 7 bits for the corresponding function field. 6 different branch operations are supported, which leads to 3 bits for the function field in B-type instructions. I-type and J-type instructions can perform only one operation respectively. Hence, no function field is required for them. At this design stage, the immediate field of R-type instructions is used for SPM addresses only. Because SPM can be chosen arbitrarily large, this immediate field has been decided to cover all remaining bits after having determined other field sizes. SPM size will be defined according to the available bits for this field then. Remaining unconstrained fields are:

- Result instruction offset
- Immediate field of I-type instructions (pure immediate values)
- Immediate field of J-type instructions (jump offset)
- Immediate field of B-type instructions (branch offset)

The static distributions of values for these fields are measured for both benchmark sets. A maximum result instruction offset of 255 is assumed. For immediate fields, a maximum bit width of 24 is assumed. Figure 11.1 provides the distribution of result instruction offsets. The large majority lies between 0 and 3. 5 bits are sufficient to cover 99.99 % of all offsets for SPEC benchmarks and 99.94 % for micro benchmarks. This leaves 6 bits for the immediate field in R-type instructions.

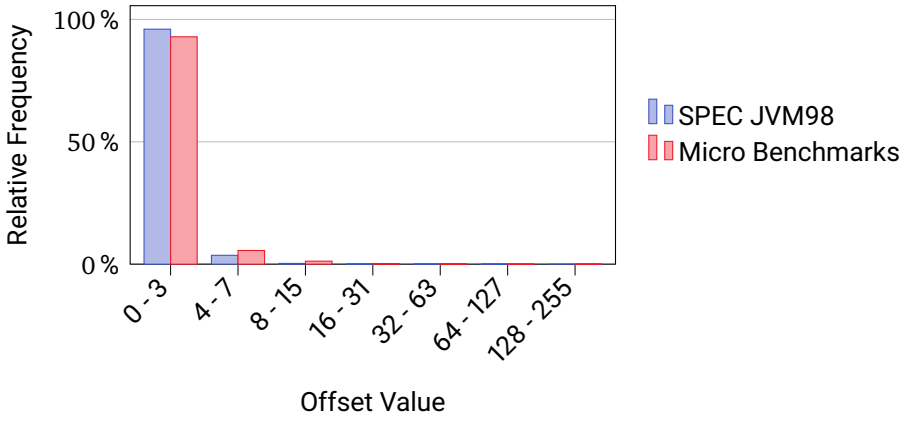


Figure 11.1.: Distribution of result instruction offsets

Figures 11.2 and 11.3 show the distributions for pure immediate values, branch offsets, and jump offsets. Results are similar for both benchmark sets. For SPEC benchmarks, 46.7 % of pure immediate values lie in the interval [0,7], 53.1 % for micro benchmarks. There is a clear bias towards positive values. If all remaining 14 bits in I-type instructions are used for symmetric encoding of immediate values, 98.9 %/96.1 % of them can be covered. Coverage can be increased to 98.3 % for micro benchmarks by introducing an offset of $6144 = 3 \cdot 2^{11}$. This offset requires an additional 3 bit adder with one constant operand. It can be implemented using three 5-input lookup tables of which two can be packed together because of identical inputs. As the adder does not lie on a critical path, hardware costs are negligibly small. Therefore 14 bits with an offset of 6144 are chosen for this field.

All branch and jump offsets can be covered by 15 bits. Since there is currently no other use for the remaining bits in J-type and B-type instructions, they can be spent for branch and jump offsets completely. This avoids problems with larger methods. Consequently,

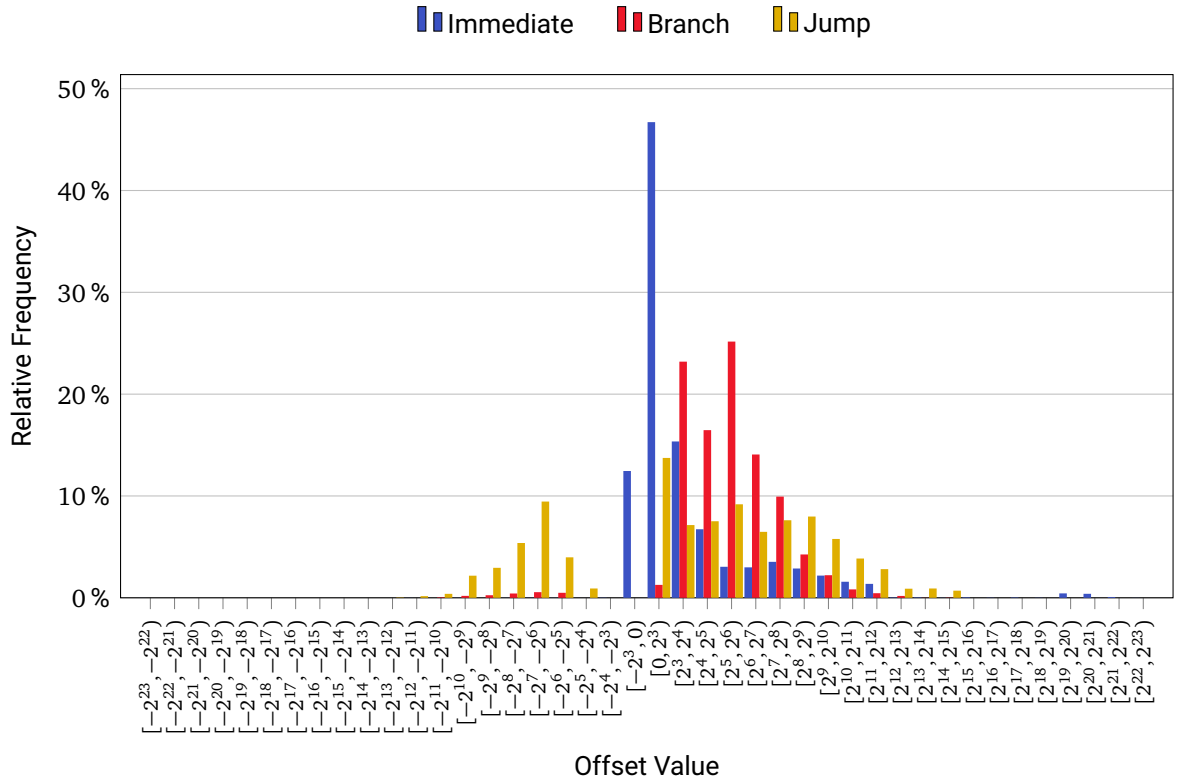


Figure 11.2.: Distribution of pure immediate values, branch offsets, and jump offsets for SPEC JVM98 benchmarks

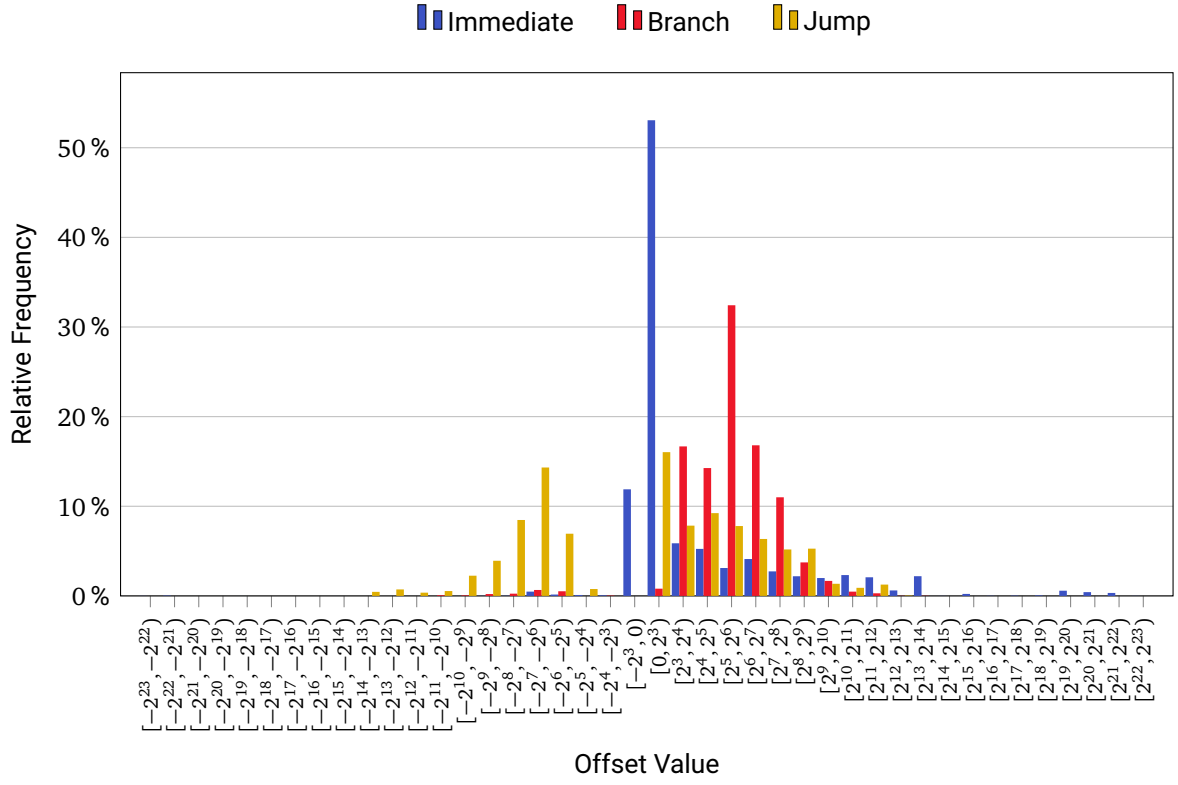


Figure 11.3.: Distribution of pure immediate values, branch offsets, and jump offsets for micro benchmarks

the immediate field in J-type instructions comprises 21 bits, the immediate field in B-type instructions 18 bits. The final binary encoding is illustrated in fig. 6.7.

11.2. Calibration of HDL Simulation

Behavioral HDL simulation is used to evaluate the performance impact of parameters quickly. In order to achieve valid results, it is necessary to confirm equivalence of HDL simulation with hardware implementation. HDL simulation is completely clock cycle accurate except for the external DRAM and its controller. They are replaced by a simplified simulation model whose AXI interface timing parameters are measured on hardware with an interface clock rate of 100 MHz. Average parameter values are given in table 11.2.

Delay between...	**Clock Cycles**
...accepting two read address transfers	1
...read address transfer and first read data transfer	27
...read data transfers of same burst	0
...accepting two write address transfers	1
...write address transfer and accepting first write data transfer	2
...accepting write data transfers of same burst	0
...last write data transfer and write response	2

Table 11.2.: AXI interface timing parameters of the simulation model for external memory

After setting model parameters, all micro benchmarks are both simulated and executed on hardware. The relative errors of simulated clock cycles compared to hardware execution are shown in fig. 11.4. The relative root mean square error is $9.53 \cdot 10^{-4}$. The worst case relative error is $2.24 \cdot 10^{-3} < 1\,\%$. This proves the high accuracy of HDL simulations for performance measurement.

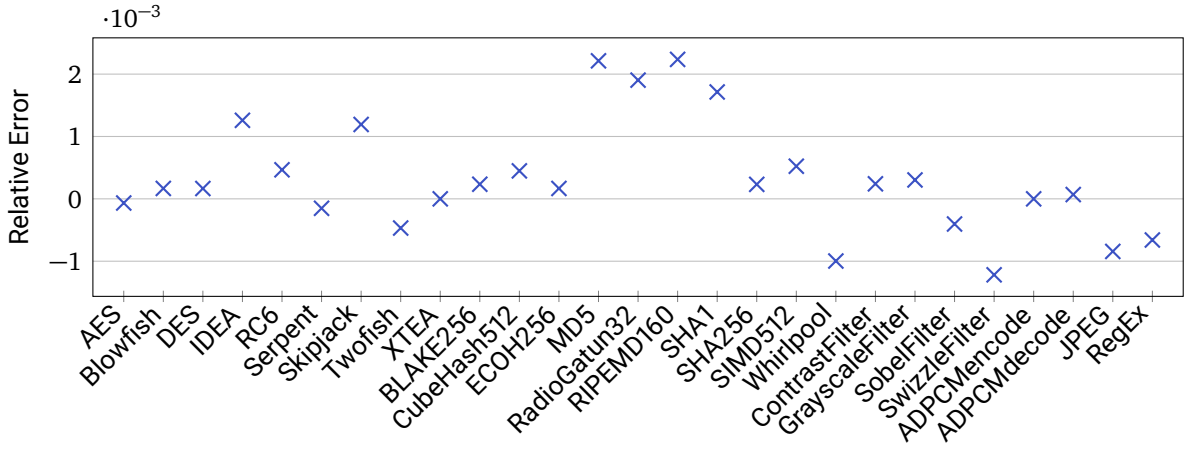


Figure 11.4.: Relative error of simulated clock cycles in comparison to execution on hardware

11.3. Scratch Pad Memory

This section has already been published in [9] with older measurements. SPM is a small but fast memory for replicating data (see section 6.1). It can be realized as a dedicated FU. However, LV slots of the Frame Stack can be used for the same purpose. Access to LVs is not much slower than access to SPM (2 vs. 1 clock cycles for reading). Figure 11.5 shows average speedup for both variants from 4 to 64 storage slots. The number of storage slots is limited by the 6 bit immediate field in R-type instructions. The geometric mean over all micro benchmarks has been calculated for each point. Figure 11.6 depicts corresponding resource consumption of the whole system. Only 6-input lookup tables (LUTs), flip-flops (FFs), and RAM blocks (BRAMs) are shown. The number of DSP blocks remains constant for all variants. All values are normalized to the variant with dedicated FU and 4 slots.

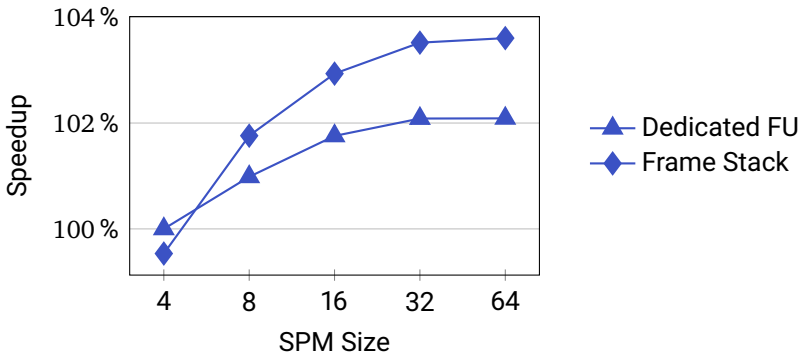


Figure 11.5.: Normalized speedup for SPM variants

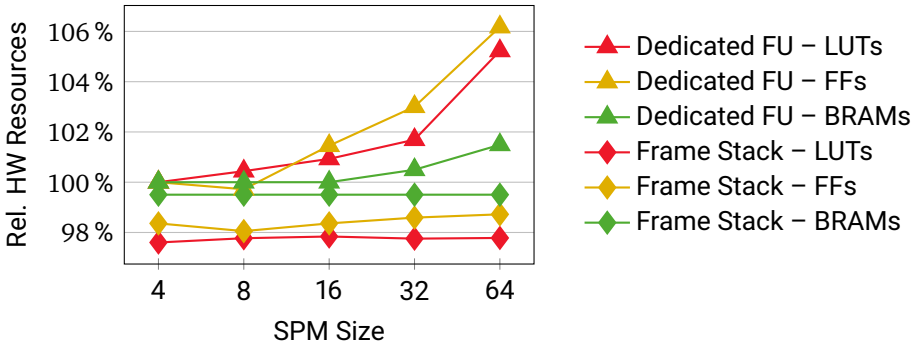


Figure 11.6.: Normalized hardware resources for SPM variants

The variant without dedicated FU exhibits significantly higher performance and consumes less resources. Increased performance can be explained by the fact that SPM contents must be saved on method invocations. This is not required for LVs because the Frame Stack organizes its memory as stack. Since hardware resource consumption is almost constant for the combined variant, the maximum number of slots is chosen for further evaluation.

11.4. Data Interconnect

The data interconnect provides a large scope for implementation alternatives. As explained in section 7.1, a topology with multiple parallel buses is applied where each FU can send to all buses and can receive from one bus. This leaves two degrees of freedom. The number of buses can be varied, which determines the amount of parallelism allowed by the interconnect. Furthermore, the assignment of FUs to buses for receiving data must be defined.

Consequently, the first step to an optimized data interconnect is to examine how the amount of buses influences application performance and hardware costs. Figure 11.7 illustrates speedups and required hardware resources for various numbers of buses. All values have been normalized to the variant with a single bus. Geometric means over all micro benchmarks yield the speedup points. RAM block usage and DSP block usage are not shown because they remain constant. The variant with 12 buses is equivalent to a fully connected crossbar. No LUT and FF numbers are provided for this variant because it fails to meet timing constraints. Speedup hardly increases with more than two buses. Therefore, this topology is chosen for further evaluation. Measuring the number of parallel data transfers per clock cycle for the 12 bus variant reveals the reason why two buses are sufficient. Figure 11.8 shows the corresponding distribution considering all micro benchmarks. No data is transferred in 67.8 % of all clock cycles. More than two transfers are almost never performed simultaneously.

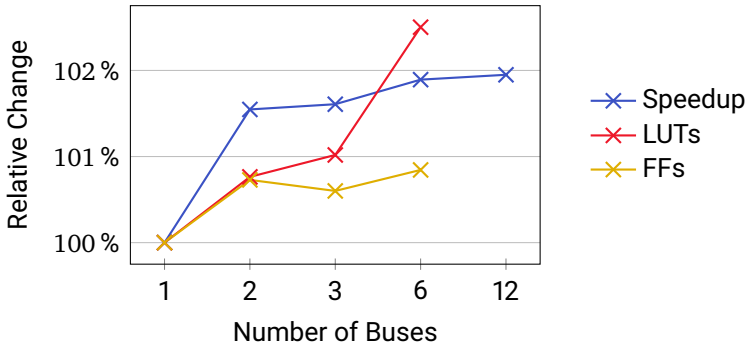


Figure 11.7.: Normalized speedup and hardware resources for number of buses

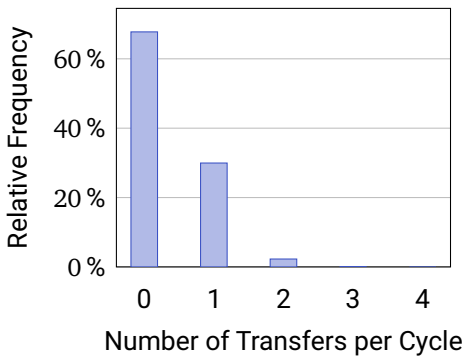


Figure 11.8.: Distribution of data transfers per clock cycle

The second step is to assign FUs to the two buses. The Debugger FU is neglected in the following discussions as it is not used during normal program execution. In general, it is

advisable to assign an equal number of FUs to each bus because this reduces the critical path of the data interconnect to a minimum. Furthermore, the number of bus conflicts should be minimized by the assignment. Two strategies are tested.

The first strategy tries to minimize bus conflicts by distributing transmission targets to the two buses equally. This is a reasonable approach because bus conflicts occur only if the targets of two transfers are assigned to the same bus. Figure 11.9 depicts the distribution of bus transfer sources and targets summarized over all micro benchmarks. Frame Stack and Token Machine are the main sources of data. Heap, Integer ALU, and Frame Stack are the main targets of data.

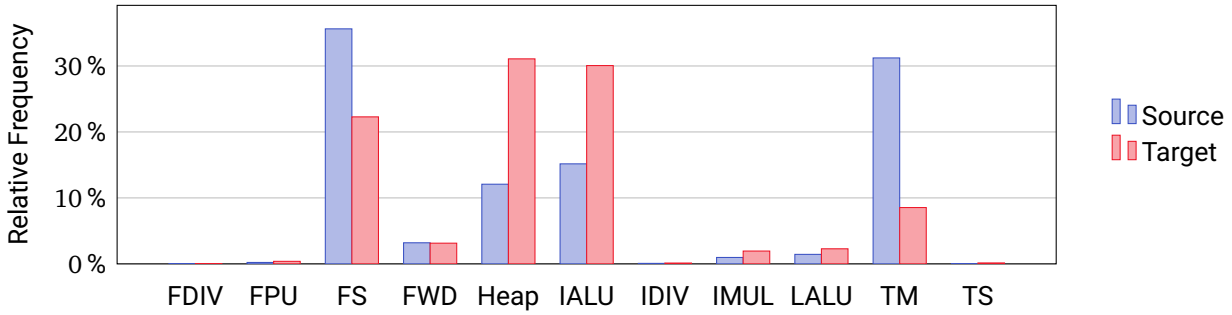


Figure 11.9.: Distribution of bus transfer sources and targets

The second strategy is based on the measurement of conflicts which would occur if only a single bus was used. Table 11.3 shows the relative numbers of conflicts for each pair of target FUs. By far the most conflicts occur between transfers which are sent to identical FUs. Unfortunately, these conflicts cannot be avoided by changing the bus assignment. Other pairs with frequent conflicts are IALU/Heap, Heap/FS, and IALU/FS. Such pairs are tried to be split to separate buses, which is not possible for all pairs with only two buses.

	FDIV	FPU	FS	FWD	Heap	IALU	IDIV	IMUL	LALU	TM	TS
FDIV	0 %										
FPU	0 %	0.1 %									
FS	0 %	0 %	3.7 %								
FWD	0 %	0 %	0.1 %	0 %							
Heap	0 %	0.1 %	3.5 %	0.4 %	26.7 %						
IALU	0 %	0 %	2.5 %	0.2 %	6.2 %	30.1 %					
IDIV	0 %	0 %	0 %	0 %	0 %	0.1 %	0 %				
IMUL	0 %	0 %	0.1 %	0 %	1 %	1.5 %	0 %	0.8 %			
LALU	0 %	0 %	0.1 %	0 %	0.2 %	0 %	0 %	0 %	2.5 %		
TM	0 %	0 %	0.6 %	0 %	0.8 %	0.6 %	0 %	0 %	0 %	0 %	
TS	0 %	0 %	0 %	0 %	0 %	0 %	0 %	0 %	0 %	0 %	0 %

Table 11.3.: Bus conflicts between pairs of target FUs

The resulting assignments for both strategies are given in table 11.4. The baseline assignment has evolved unintentionally during development. The corresponding speedups are depicted in fig. 11.10. Obviously, the assignment of FUs to buses hardly has an impact on performance. The baseline is the fastest variant on average. Consequently, it is selected for further study.

	FDIV	FPU	FS	FWD	Heap	IALU	IDIV	IMUL	LALU	TM	TS	DBG
Baseline	●	●	●	●	●	●	●	●	●	●	●	●
Equal Target Distribution	●	●	●	●	●	●	●	●	●	●	●	●
Split Target Conflicts	●	●	●	●	●	●	●	●	●	●	●	●

Table 11.4.: Variants for assigning FUs to data buses (color indicates bus)

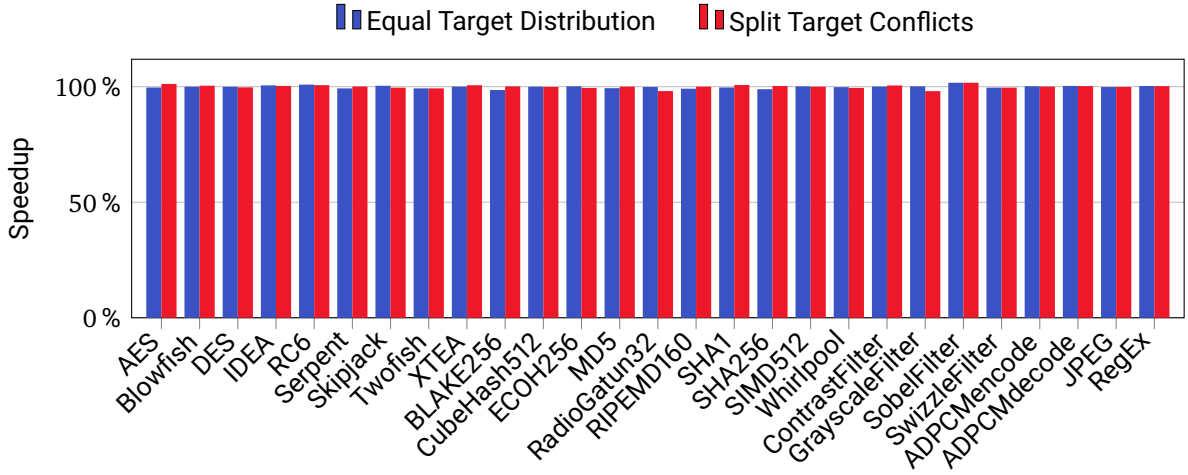


Figure 11.10.: Speedup for bus assignment variants compared to baseline assignment

11.5. ISA Parameters

Section 7.3 names several hardware parameters which are relevant for programming the system and hence must be considered part of the ISA. These parameters are resolution capacity, operation memory sizes, and result queue sizes. A sweep over the parameter values is conducted to find the optimal configuration. At first, the parameters of all operation categories are equal. The resolution capacity is evaluated between 8 and 32. The lower bound has been found experimentally. With a resolution size of 4, it is hardly possible to implement programs without storing each intermediate value into LVs. The upper bound is given by the bit width of the instruction offset field in binary encoding. The operation memory size is varied between 4 and 32. A lower value limits the capability of receiving operands out-of-order severely. Operation memories which are larger than the resolution capacity are not useful either. The result queue size is evaluated between 2 and 32. At least double buffering of the output should be supported. Again, exceeding the resolution capacity is not beneficial.

An excerpt of the design space is illustrated in fig. 11.11. All plots contain the baseline configuration defined in table 11.1. Speedup and hardware resources are normalized to this configuration. Speedup points represent the geometric means over all micro benchmarks. The number of failed benchmarks is given by absolute figures. All failures occur during instruction scheduling. Every schedule found for some configuration is also valid for all configurations with higher ISA parameters. Consequently, the number of failed benchmarks should be monotonically decreasing. Obviously, scheduling fails

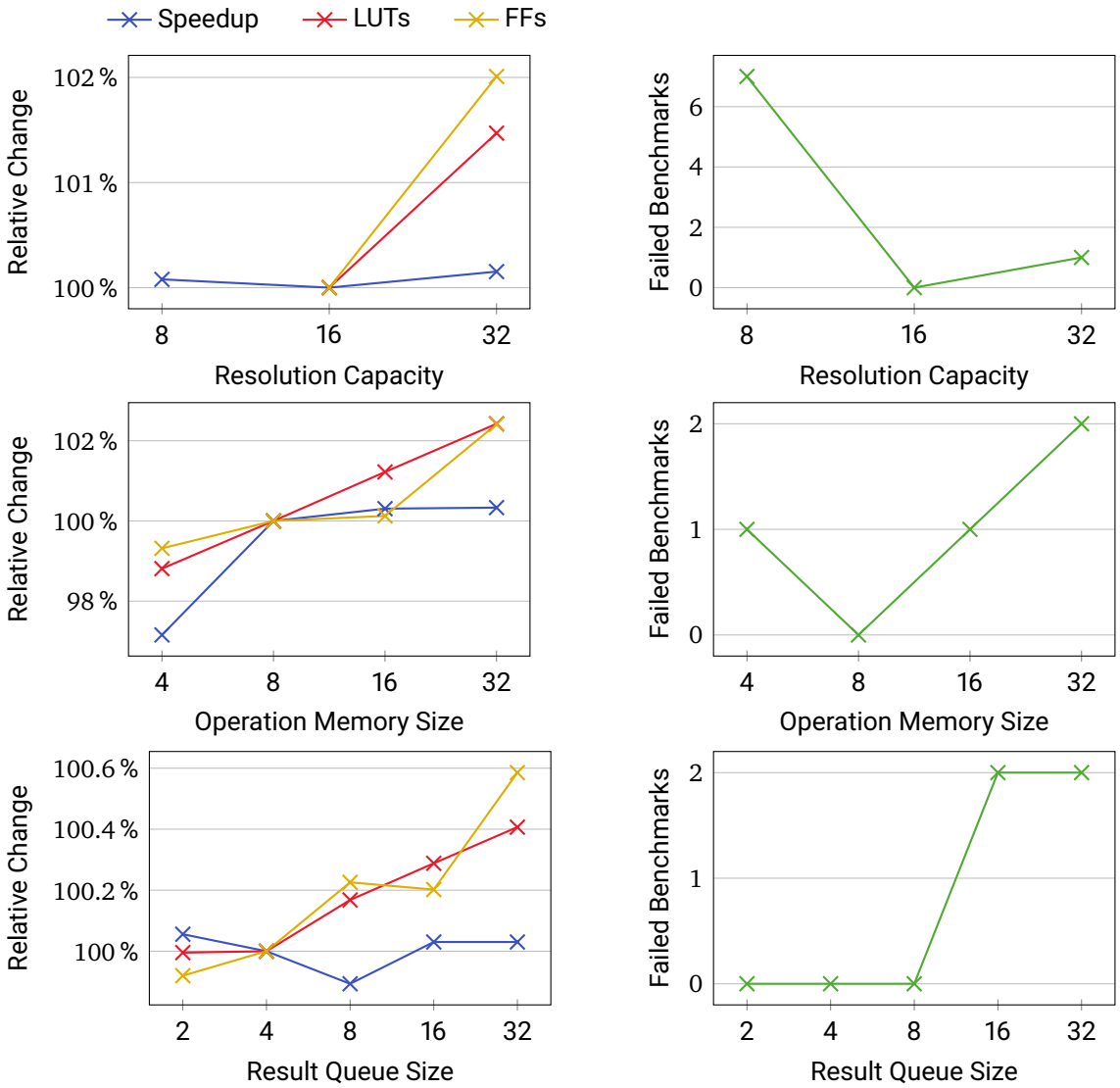


Figure 11.11.: Speedup, hardware resources, and failed benchmarks for different ISA parameter values. Speedup and hardware resources have been normalized to the baseline configuration.

to find existing, valid schedules. The same two benchmarks fail for high parameter values: *BLAKE256* and *RadioGatun32*. They stand out due to large loop bodies with much ILP and leave the scheduler plenty of room for choices. The constraint checks tend to exhaust limits. Thereby, their decisions to satisfy one constraint can make handling of other constraints more difficult or even infeasible. Furthermore, deadlocks are resolved individually. Resolving one deadlock can hamper the resolution of other deadlocks or can even introduce new ones. Considering multiple deadlocks or constraints for each decision would be beneficial but rather complicated. Avoiding exhausted constraint limits could help as well but would lead to wasted resources. The most promising measure is to support shifting of problematic direct values to LV storage during preliminary instruction scheduling. For both *BLAKE256* and *RadioGatun32*, such problematic values are identified quickly after few scheduling iterations. The scheduler tries to insert forwards repeatedly, which does not lead to a solution. It would be possible to detect these situations and to

transfer the values as LVs instead. However, this approach is not possible during final scheduling, which takes place after allocation of LV slots.

In general, variations of performance and hardware are low inside the design space. The best performing configuration lies at 100.45 % speedup, 101,72 % LUTs, and 100,48 % FFs. The configuration with lowest hardware requirements lies at 97.23 % speedup, 98,92 % LUTs, and 99,50 % FFs.

Resolution Capacity Too many benchmarks fail with a resolution capacity of 8. Consequently, this value has been excluded from the synthesis sweep. Furthermore, resolution capacity has no significant impact on performance while hardware requirements increase. Therefore, 16 is identified as ideal value.

Operation Memory Size Performance increases from operation memory sizes 4 to 8 but hardly for higher values. Since hardware requirements increase monotonically, 8 is selected as optimal value.

Result Queue Size Both performance and hardware resource requirements are similar for result queue sizes 2 and 4. With higher values, performance remains almost constant while hardware requirements increase. Therefore, 2 and 4 can be seen as optimal values. 4 is chosen to leave room for further applications.

Finally, operation memory and result queue sizes are tried to be reduced for individual operation categories without impacting performance or causing benchmark failures. All FUs within an operation category must be treated equally. As result, operation memory sizes are reduced to 4 and result queue sizes are reduced to 2 for Integer Divider, Floating Point Divider, Thread Scheduler, and Debugger.

11.6. Compact Code

If the execution of DOJA programs is examined, several weaknesses can be discovered. One problem is the large number of immediate instructions. If the *Imm6* field in R-type instructions is used for more operations than `read` and `write`, many immediate instructions can be saved. Another problem is inefficient passing of method parameters. The combination of `read` and `push32` occurs very often. Both operations are executed by the Frame Stack because the dedicated SPM has been removed. Therefore, a new operation can be implemented which provides the same functionality without external data transfer. In total, 128 operations can be encoded in R-type instructions, of which 116 are occupied. Consequently, 12 additional operations can be introduced for a more efficient encoding of programs in terms of code size and performance.

Static code analysis delivers opportunities for such improvements. Table 11.5 shows the most frequent operations which could use the *Imm6* field for encoding constant operands. Results are similar for both benchmark sets. Micro benchmarks use array accesses with constant index more often. They are used mainly for array initialization. Since such initialization typically happens rarely during normal program execution, they are excluded. The operations for which equivalent immediate variants are introduced are marked in red. No new operation is necessary in case of `load32` because `read` is semantically equivalent

SPEC JVM98 Operation	Frequency	Micro Benchmarks Operation	Frequency
return_fs	3.48 %	getfieldh	3.54 %
inv_virt	3.20 %	putarray	2.17 %
load32	3.10 %	return_fs	1.70 %
inv_static	2.41 %	getfield	1.64 %
getfieldh	1.92 %	load32	1.60 %
invoke_fs	1.59 %	getarray	1.45 %
putfield	1.22 %	add	1.44 %
getfield	1.04 %	putfield	1.28 %
push32	0.53 %	inv_static	1.15 %
add	0.49 %	push32	1.08 %
cmp	0.36 %	invoke_fs	0.80 %
alloc_obj	0.20 %	inv_virt	0.77 %
i2l	0.17 %	shl	0.50 %
putarray	0.11 %	const32	0.24 %
const32	0.11 %	shra	0.23 %

Table 11.5.: Most frequent operations which could use the *Imm6* field for encoding constant operands. Red operations are complemented with an immediate version.

without dedicated SPM. In general, all `load` and `store` operations can be replaced by `read` and `write` operations as long as the address lies in the interval [-32,31]. Negative addresses are used for loading method parameters.

Frequent Frame Stack operations which could be combined are listed in table 11.6. As expected, the most frequent combination is `read` and `push32`. `write` and successive `read` of the same address occurs often as well. Hence, new operations are introduced for these combinations.

SPEC JVM98 Operation	Frequency	Micro Benchmarks Operation	Frequency
read-push32	4.98 %	read-push32	4.23 %
write-read	3.34 %	write-read	2.44 %
load32-write	1.56 %	load32-write	0.92 %
load32-push32	0.66 %	load32-push32	0.24 %
read-store32	0.02 %	read-store32	0.00 %

Table 11.6.: Most frequent operation combinations which could be replaced by single operations. New operations have been introduced for the red combinations.

Figure 11.12 depicts the reduction of code size caused by introducing additional operations. Results are very similar for all benchmarks. Code size is reduced to 77.0 % on average. Performance is affected as well, which can be seen in fig. 11.13. Run times of all benchmarks have been measured on hardware before and after implementing optional

operations. An average speedup to 107.8 % is achieved. Hardware resource consumption is not changed significantly. The number of LUTs has been increased by 0.11 %. The number of FFs has been decreased by 0.50 %. Both numbers lie within the typical range of synthesis result variations.

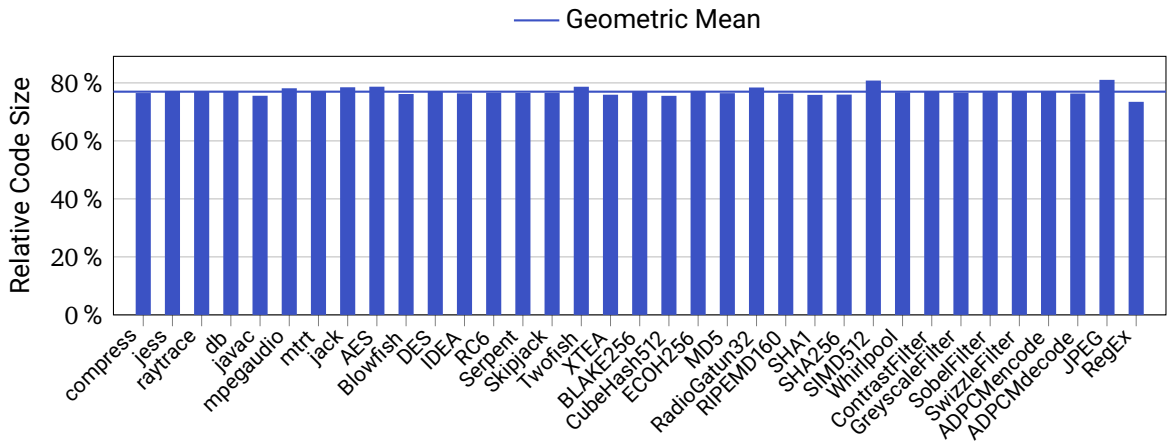


Figure 11.12.: Reduction of code size by optional operations

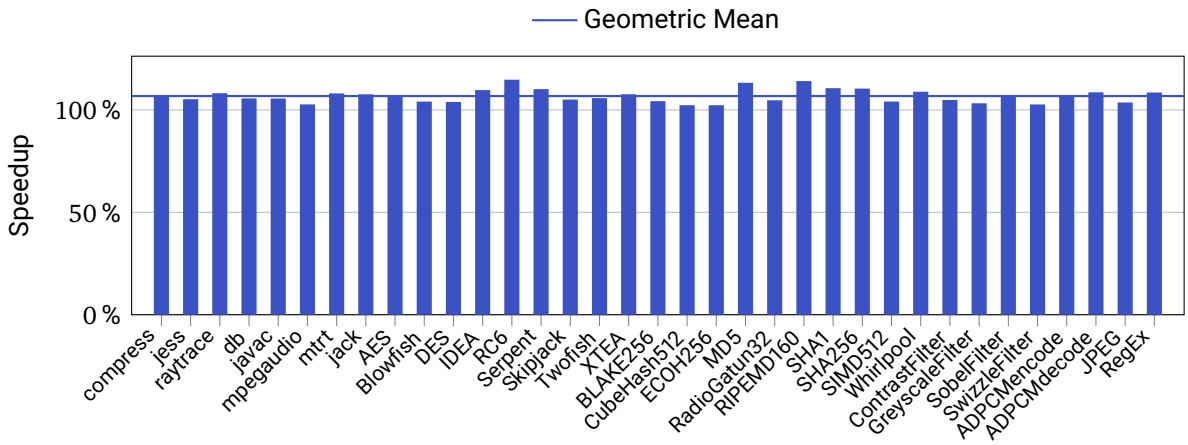


Figure 11.13.: Speedup by optional operations

12. Comparison of Architectures

After having optimized ISA and hardware implementation, the new DOJA-based system must be compared to the previous Java bytecode-based system. The same target hardware platform and the identical FPGA synthesis tool chain are used for both systems (see section 10.2 for details). Furthermore, both processor cores run at 100 MHz. Since large parts of the systems have not been changed including computational FUs, a fair comparison between the architectures can be provided. In contrast to the bytecode-based system, the Heap FU of the DOJA-based system supports copying arrays in hardware. This feature is deactivated for measurements in order to avoid distorted results.

Table 12.1 gives the values of the hardware parameters used for both systems. The first five parameters are relevant for the new system only. They have been explored in the previous chapter. Further constant parameters can be found in appendix B. All numbers in this chapter are measured on hardware. Benchmark performance is determined by measuring the run times of the corresponding main methods. Since static initializers are called before the main method, their run times are not included. Other values are investigated using performance counters which are active only during execution of the main method. This ensures consistent numbers. Performance counters are not included in hardware resource consumption.

Parameter	**Value**
Dedicated SPM Addresses	✗
Number of Data Buses	2
Operation Memory Sizes	4 for IDIV, FDIV, TS, DBG 8 for all other FUs
Result Queue Sizes	2 for IDIV, FDIV, TS, DBG 4 for all other FUs
Resolution Capacity	16
Max. Number of Live Threads	16
Max. Number of Acquired Monitors	64
Stack Addresses per Thread	4096

Table 12.1.: Hardware parameters of the evaluated systems

12.1. Performance

Benchmark performance is the most important metric for assessing the new architecture. Figure 12.1 shows achieved speedups, which are 1.87 on average for SPEC JVM98 benchmarks and 2.89 for micro benchmarks. The highest speedup of 4.11 can be observed with

GrayscaleFilter. Hence, DOJA provides significant gain compared to Java bytecode. The notable difference between the two benchmark sets is caused by much more complex control flow with many method invocations in SPEC benchmarks. Consequently, DOJA is beneficial especially for large, computation-intensive methods.

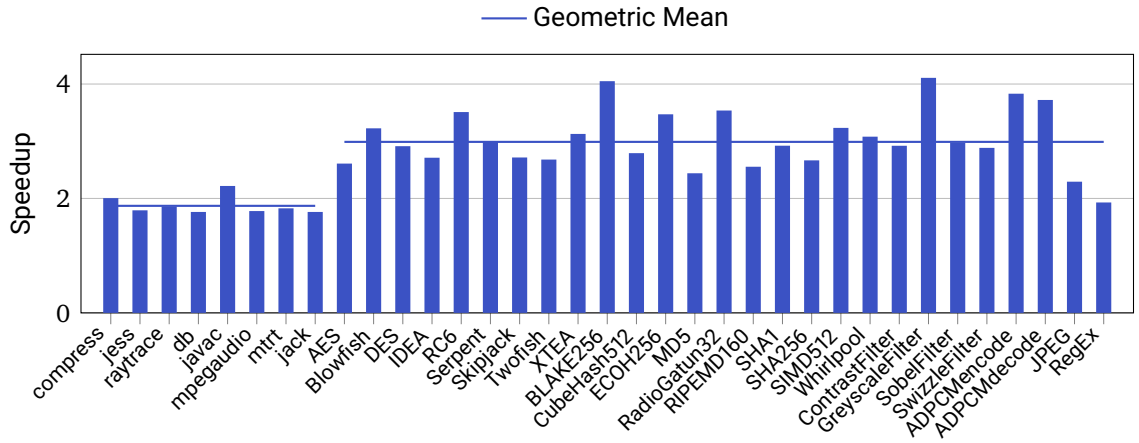


Figure 12.1.: Speedup in comparison to Java bytecode

Speedup is achieved mainly by elimination of unnecessary FU operations, which is illustrated in fig. 12.2. The total amount of FU operations has been reduced to 52.5 % for SPEC benchmarks and to 45.2 % for micro benchmarks. Benchmarks with a lower reduction invoke very small methods frequently (e.g. *raytrace*, *mtrt*, *SHA256*) or make excessive use of exception handling (e.g. *jack*). Exception handling in software causes additional FU operations for the new processor.

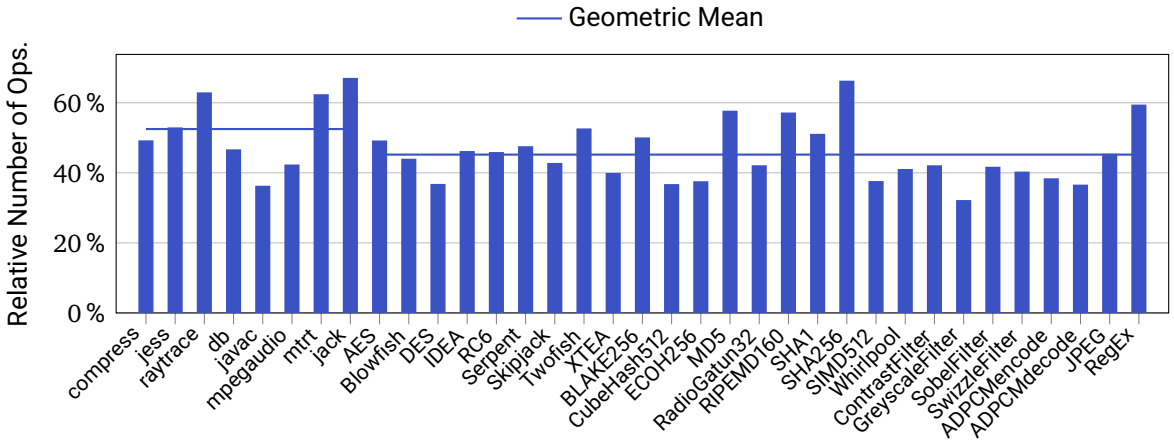


Figure 12.2.: Number of executed FU operations in comparison to Java bytecode

Figure 12.3 depicts the number of executed operations per FU and the number of data transfers originating from each FU. Values are summarized over all benchmarks. Operations are reduced mainly for the Frame Stack (by 61.9 %) and for the Token Machine (by 68.2 %). Reduction of Frame Stack operations has been the major design goal of the new ISA. Token Machine operations are reduced primarily because less constants must be sent over the data interconnect. This has two reasons. First, reducing LV usage means

reducing constants for LV addresses. Second, many small constants are transferred to FUs via TDN along with operations. The Integer ALU performs less operations especially because comparisons with zero for branches are executed directly by the Token Machine. The Forward FU of the new processor is used rarely.

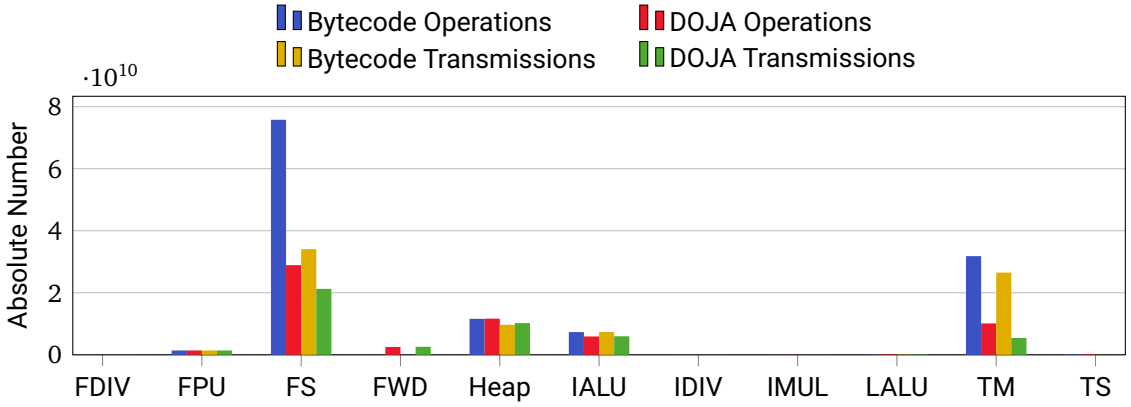


Figure 12.3.: Number of FU operations and of data transfers summarized over all benchmarks

The reduction of data transfers originating from the Frame Stack (by 37.6 %) is lower than the reduction of Frame Stack operations. This can be explained by many `write` and `push` operations, which do not produce results. For the Token Machine, the reduction of data transfers (by 79.5 %) is even higher than the reduction of operations because mainly `imm` operations have been eliminated as stated above.

Further speedup is achieved by allowing more operations to be processed concurrently because not all intermediate data must be stored in the Frame Stack. Unfortunately, this claim is difficult to prove directly with measured numbers. The average amount of PEs per clock cycle which are processing operations simultaneously is actually reduced for the new architecture because of the much lower number of operations in total. Additional performance is gained from the higher throughput of the operand matching mechanism (one data transfer every clock cycle instead of every second cycle).

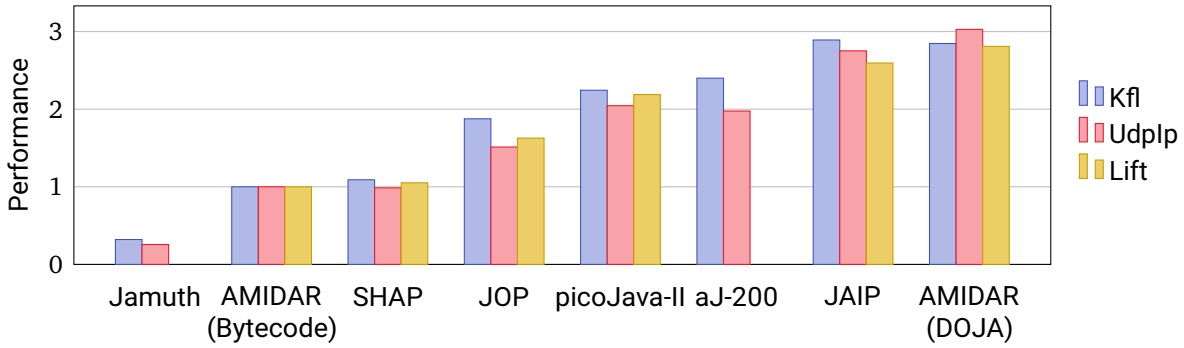


Figure 12.4.: JemBench application benchmark performance relative to bytecode-based AMIDAR [32, 36]

Finally, a rough comparison to other Java processors can be drawn when looking at JemBench [37] application benchmarks in fig. 12.4. The DOJA-based AMIDAR processor

is the fastest one in this comparison (Kfl: 30200, UdpIp: 17693, Lift: 32669). It must be noted again that this benchmark set does not even closely exhaust the capabilities of the AMIDAR system. It is used here only because benchmark scores of other processors are provided in literature. CGRA acceleration does not provide significant speedup for these benchmarks.

12.1.1. Multithreading

Since *mtrt* is a multithreaded variant of *raytrace*, the influence of the more sophisticated context switching mechanism on performance can be examined. Speedups of *mtrt* and *raytrace* differ only slightly. Consequently, the change in context switching hardly impacts performance.

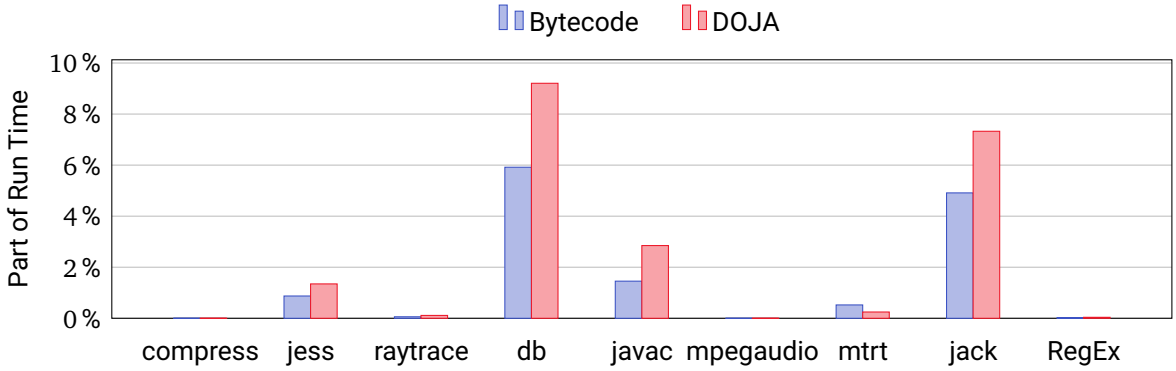


Figure 12.5.: Part of run time spent for thread switch requests

Figure 12.5 provides deeper insight into thread management overhead. It shows the part of run time which is spent for thread switch requests from stopping to restarting the decoding pipeline. Not every request leads to a real thread switch, which is often the case for `monitorenter` instructions. The only benchmark which switches between threads cyclically is *mtrt*. Other SPEC benchmarks switch to the UART IST temporarily for sending standard output. *RegEx* provokes thread switch requests but no thread switches. All other benchmarks do not require thread switch requests at all. Hence, they are excluded from the diagram. The figure reveals that some benchmarks spend significant amount of time for thread switch requests, especially *db* and *jack*. Since UART output happens rarely, this overhead must be caused by frequent use of thread synchronization. Relative numbers mostly are higher for DOJA because the rest of the program is executed faster, not because thread switch requests are slower.

This becomes clear when looking at fig. 12.6, which illustrates the average number of clock cycles per thread switch request. Numbers are similar for Java bytecode and DOJA. Moreover, the average number of clock cycles for pure thread switches without the time required by the Thread Scheduler is shown for the DOJA processor. In comparison, the Java bytecode processor requires exactly one cycle for sending a thread switch operation to the Frame Stack. Even for *mtrt*, only 14 cycles are spent on average. Figure 12.7 reveals the reason. Usually, only one unresolved target with corresponding data word must saved when switching threads. Consequently, the thread switch overhead introduced by DOJA is negligible. The major part of time for thread switch request handling is consumed by the Thread Scheduler.

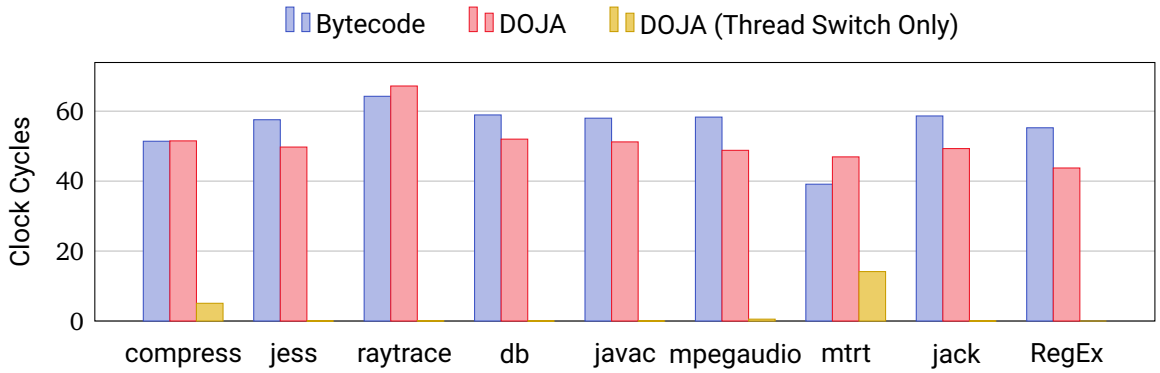


Figure 12.6.: Clock cycles per thread switch request

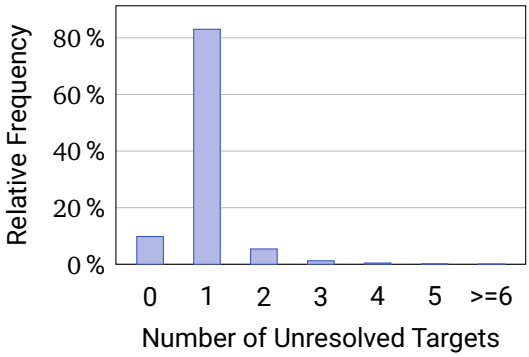


Figure 12.7.: Distribution of unresolved targets per thread switch in *mtrt*

12.1.2. Instruction Processing

In order to gain ideas for future improvements, it is worth examining instruction processing in more detail. Figure 12.8 depicts the relative amount of time spent in each decoding pipeline state. Numbers are the arithmetic means over all SPEC JVM98 benchmarks and all micro benchmarks respectively. Normal instruction processing consumes 21.5 % of the time for SPEC benchmarks and 47.9 % for micro benchmarks. This part of run time could be reduced by faster instruction decoding. A dual issue pipeline seems to be worthwhile at least for micro benchmarks.

Full operation memories or full target queues block the pipeline in 15.0 % and 13.8 % of the clock cycles. FU processing is slower than instruction decoding in these cases. Measurements for individual FUs are necessary to identify which FUs could be improved or duplicated.

The pipeline is waiting for instructions in 9.6 % and 11.7 % of the time. This could be caused by the instruction cache. Figure 12.9 shows its hit rates. A hit rate of nearly 100 % is achieved for SPEC benchmarks and 98.2 % for micro benchmarks. Consequently, the instruction cache can be eliminated from the set of possible causes. A look at the hit rate of the branch prediction unveils that its performance is not satisfying. Average hit rates of 55.3 % and 64.4 % are achieved. Values vary greatly for micro benchmarks. No target prediction exists for invocations and returns, which causes additional waiting cycles. Therefore, improving branch prediction as well as introducing invocation target prediction could reduce this waiting time.

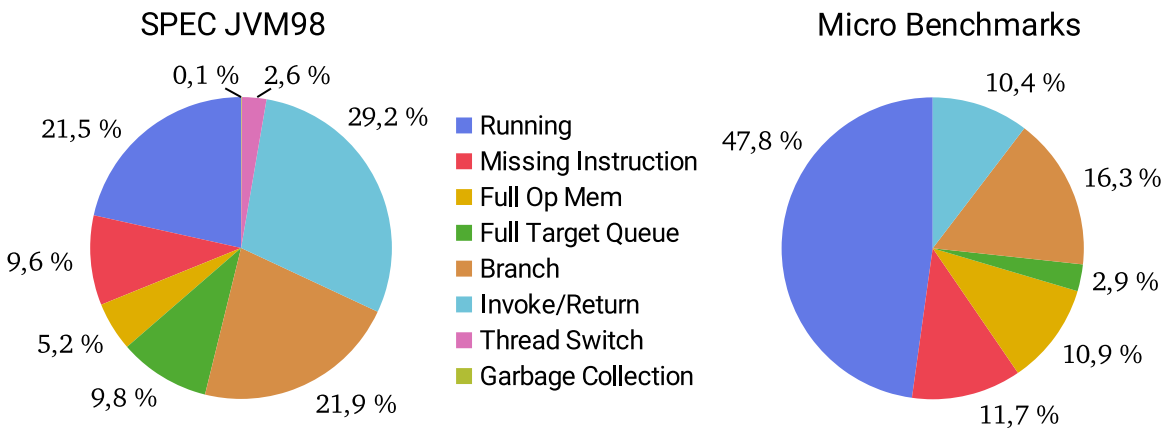


Figure 12.8.: Decoding pipeline state

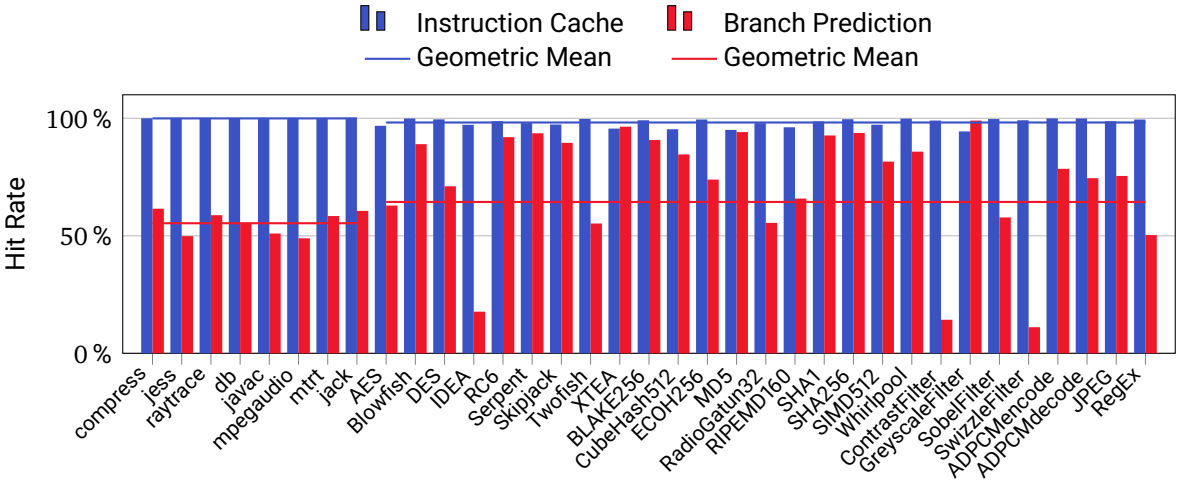


Figure 12.9.: Hit rates of instruction cache and branch prediction

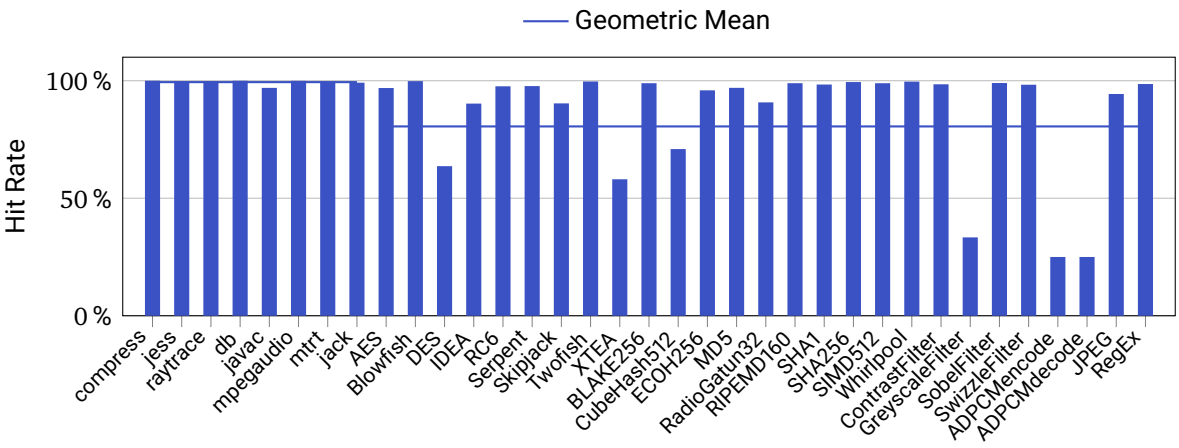


Figure 12.10.: Hit rate of method table cache

For SPEC benchmarks, the pipeline spends most of the time waiting for invocation/return targets and for branch decisions. Invocation delays could be due to bad performing table caches. However, fig. 12.10 shows method table cache hit rates near 100 % for SPEC benchmarks. Hit rates for micro benchmarks are lower but mainly because some benchmarks invoke very few methods, most of them only once. Other relevant table caches show very similar results. Therefore, lost cycles must be caused by waiting for data from other FUs (CTI from Heap or return address from Frame Stack). Similarly, branch delays must be caused by waiting for comparison data. No easy solution exists here. Detailed analysis of FU operation execution is necessary to identify the sources of delays. Additionally, support for speculative execution could help to hide waiting cycles. Of course, this problem could also be tackled at transpiler level by inlining method calls.

12.2. Hardware Resource Usage

Parts of this section have been published previously with older numbers in [9]. Figure 12.11 illustrates the amount of FPGA resources consumed by the bytecode-based and the DOJA-based systems. In total, the number of LUTs is increased by 4.8 %, the number of FFs is decreased by 8.1 %, and RAM block usage is lowered by 4.6 %. One DSP block has been removed from the Token Machine. In short, hardware resource requirements are reduced slightly.

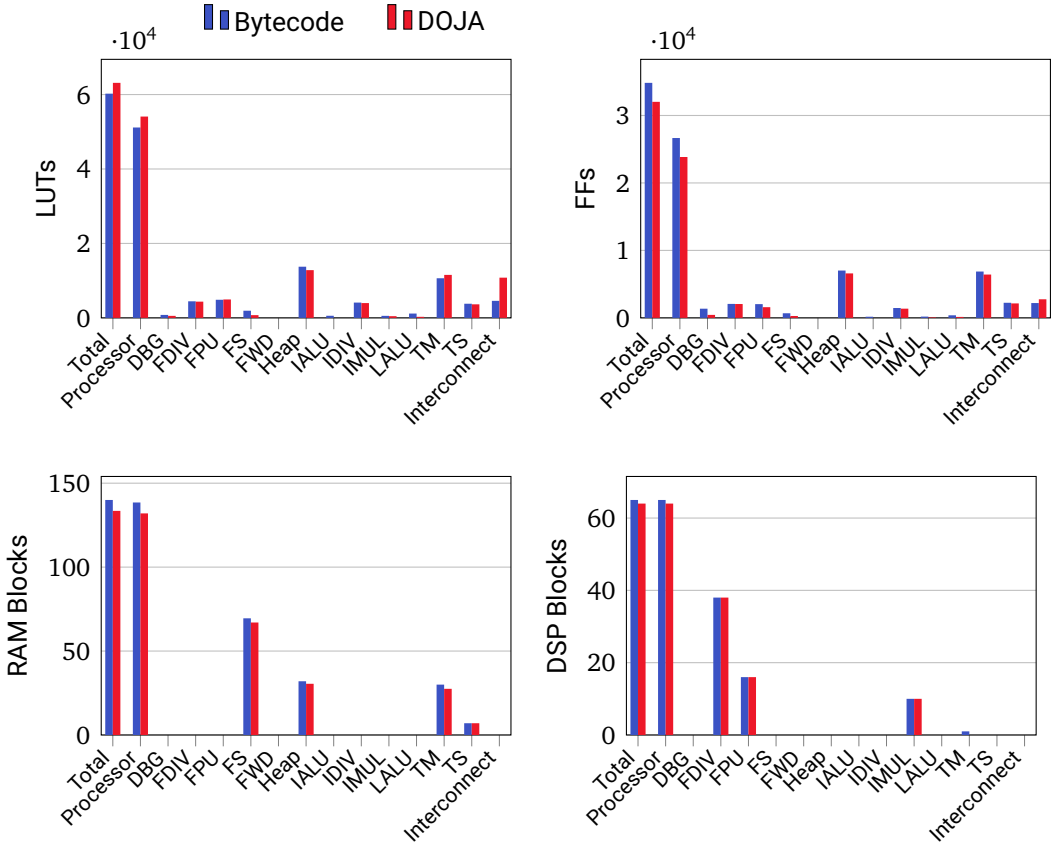


Figure 12.11.: FPGA resources consumed by both systems

The numbers for individual components of the systems are given as well. However, these numbers are not exact because optimization across module boundaries has been enabled during synthesis. In addition, some components which have been part of the FUs in the old processor are now part of the interconnect. “Processor” comprises only the resources of the processor core without peripherals, AXI interconnect, and DRAM controller. “Interconnect” includes data interconnect and TDN. The most significant change is the increase of LUTs for the interconnect. It is caused mainly by operation memories and queues, which are realized using distributed RAM.

The Wishbone interconnect with all peripherals is clocked at 25 MHz. Because some FUs have Wishbone interfaces as well, the processor core can only be clocked at multiples of 25 MHz without special measures for clock domain crossing. 100 MHz is the highest multiple which can be reached by the design on the chosen target FPGA. The critical path varies between synthesis runs if synthesis inputs are changed slightly. Typical candidates are the last pipeline stage of the FPU, the data cache, or the data interconnect from target queue to operation memory.

12.3. Code Size

The major disadvantage of the new architecture is the increase of code size compared to Java bytecode. Exact numbers are given in fig. 12.12. Code size increases by a factor of 2.82 for SPEC JVM98 benchmarks and by a factor of 2.53 for micro benchmarks. Dead code elimination is disabled for these measurements. Furthermore, only methods which exist in both program images are taken into account. Hence, elimination of unreachable methods has no influence on results.

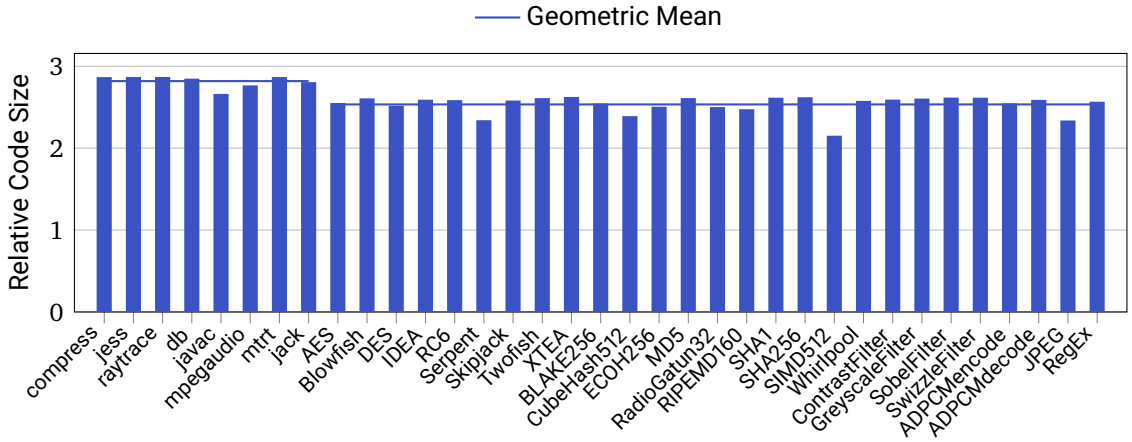


Figure 12.12.: Code size in comparison to Java bytecode

Such numbers have been expected because compactness is a strength of Java bytecode. Its dynamic average instruction size is 1.8 bytes [6] whereas DOJA has a fixed instruction size of 3 bytes. Some additional instructions are required like `fwd` or `invoke_fs`. Furthermore, Java bytecode can store relatively large constants in instructions of variable length. In contrast, DOJA requires separate `imm` instructions more often.

Program image size can be reduced drastically by removing unreachable methods, fields, and classes. Initialization of static primitive arrays can be done at transpilation time to

eliminate more code. However, these measures are not specific for the ISA and could be applied to bytecode-based AXT images as well.

12.4. Code Conversion

Conversion of Java class files to NAX images containing DOJA code is much more complex than conversion to AXT images containing Java bytecode. However, conversion times are similar for most benchmarks as depicted in fig. 12.13. Average times of 11.96 s for SPEC JVM98 benchmarks and 6.24 s for micro benchmarks are acceptable. Conversion time scales with code size for most benchmarks. *RIPEMD160* and *SIMD512* are exceptions with much larger times of up to 22.37 s. Each of both contains one very large method with much ILP. Conversion into SSA form as well as CDFG/CFG transformations consume significantly more time for these methods.

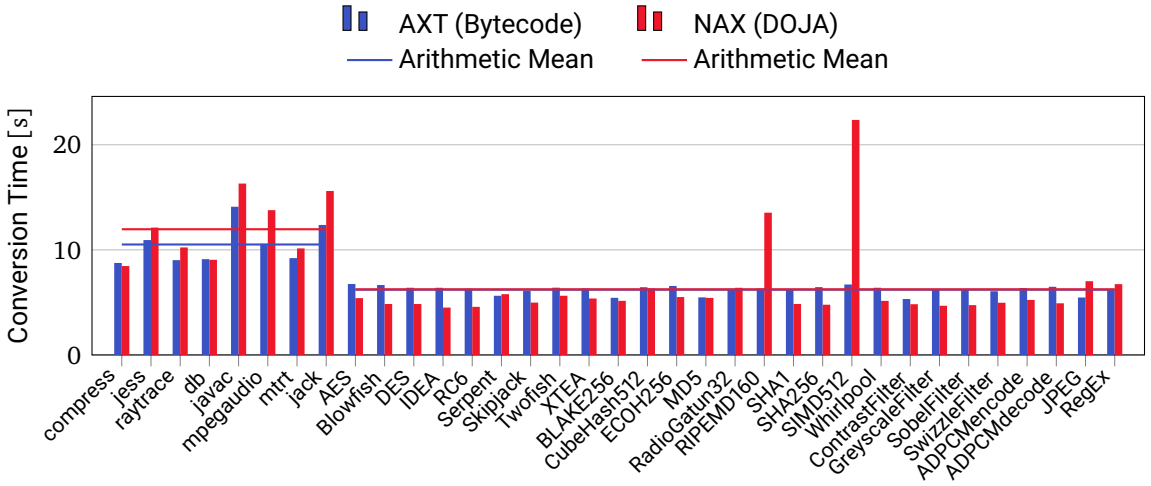


Figure 12.13.: Time for converting Java class files into AXT (Bytecode) and NAX (DOJA) images

Figure 12.14 breaks down transpilation time into its components. Numbers represent the arithmetic means over all benchmarks. Approximately one third is consumed by loading all input files (configuration, class files, assembly files). Most of this time is spent for converting class files to Jimple representations. Other expensive processes are CDFG/CFG transformations and instruction scheduling. The final scheduling pass consumes less time than the preliminary pass because of less freedom. Jimple/Shimple transformations, instruction selection, and program image export require only a minor fraction of time.

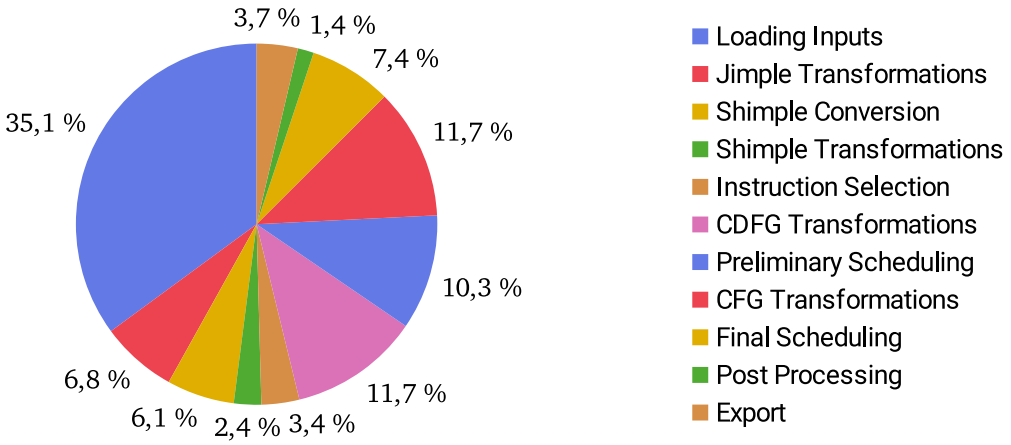


Figure 12.14.: Components of transpilation time

13. Acceleration by CGRA

A key feature of AMIDAR processors is their seamless integration of CGRAs as programmable hardware accelerators. CGRA development is still in progress. Therefore, the full potential cannot be demonstrated yet. A CGRA composition with 2x2 PEs according to fig. 2.11 is inserted into the processor on the target hardware platform described in section 10.2. CGRA memory parameters are given by table 13.1. All PEs support full integer but no floating point arithmetic. CGRA compositions with more PEs and floating point support are applicable for larger FPGAs. Multithreading parameters have been reduced in comparison to the system of chapter 12 in order to make the design fit onto the FPGA.

Parameter	**Value**
Condition Box Register File Size	64
PE Register File Size	256
Context Memory Entries	2048
Constant Memory Entries	1024
Location Information Table Entries	1024
Kernel Table Entries	32
Max. Number of Live Threads	8
Max. Number of Acquired Monitors	32
Stack Addresses per Thread	2048

Table 13.1.: CGRA memory parameters and multithreading parameters of the evaluated system

All numbers of this chapter have been measured on hardware with CGRA context synthesis executed on the development computer. Online synthesis running on the DOJA-based AMIDAR processor has been tested as well. With 1912 classes and 9086 methods, it is the most comprehensive application which is running on the processor so far. It is written in Java and Kotlin. Therefore, it proves that other programming languages besides Java are supported. Unfortunately, the L2 data cache must be removed when using online synthesis because of conflicts with garbage collection. Solving this problem is subject to future work. Another unsolved problem is synthesis speed. Online synthesis of medium sized loops takes around 30 minutes when executed in parallel to the application. Focus of synthesis software design has been high flexibility to the detriment of speed so far. Much shorter synthesis times can be expected with future improvements.

The optimal loop unrolling configuration is applied for each benchmark. These configurations, which are provided by table 13.2, have been determined experimentally. Measurements are executed only for micro benchmarks because loops suitable for CGRA context synthesis are hard to find in SPEC JVM98 benchmarks. Context synthesis does not

support recursion, polymorphism, exception handling, and object allocation, all of which frequently occur in SPEC benchmarks. 64 bit integer values are currently not supported either. Floating point computations cannot be performed by the CGRA composition as mentioned above.

Benchmark	Inner Loop	Parent of Inner Loop
DES	2	*no unrolling*
XTEA	2	*no unrolling*
MD5	2	*no unrolling*
ContrastFilter	4	2
SobelFilter	4	2
ADPCMdecode	2	*no unrolling*
all others	*no unrolling*	*no unrolling*

Table 13.2.: Optimal loop unrolling factors

13.1. Performance

Synthesizing CGRA contexts from all suitable benchmark loops generates notable speedup. The average speedup is 2.53 across all micro benchmarks. The highest speedup of 8.97 is reached for *SHA256*. This impressively shows the potential of CGRA acceleration. However, some benchmarks gain no or almost no speedup. *MD5* does not contain a loop in its core algorithm. Consequently, CGRA context synthesis does not accelerate a significant portion of the program. Synthesis of the *RIPEMD160* core loop fails because PEs do not provide enough registers. *ContrastFilter* and *JPEG* work with floating point numbers, which are not supported by the CGRA composition. *RegEx* applies object allocation.

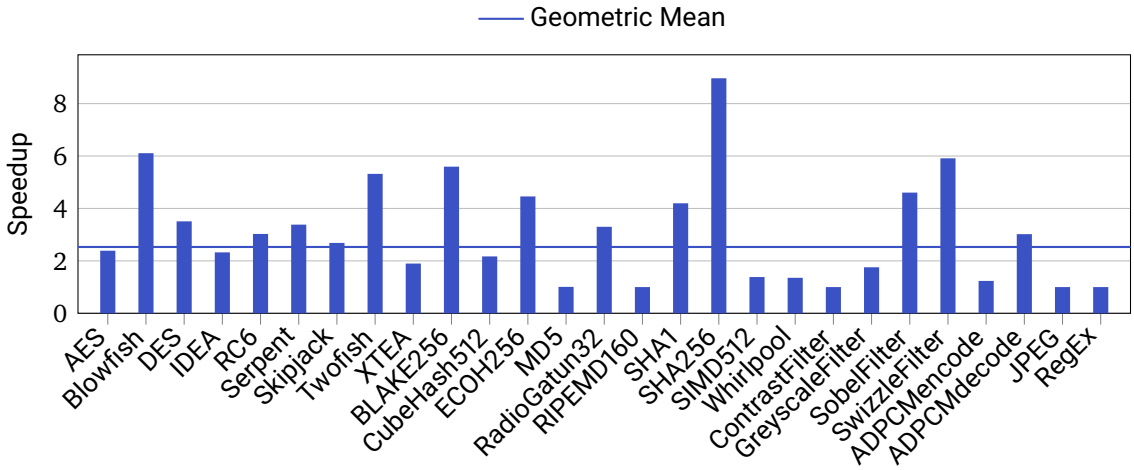


Figure 13.1.: Speedup by synthesizing CGRA contexts from benchmark loops

13.2. Hardware Resource Usage

Figure 13.2 illustrates the FPGA resources required for two AMIDAR systems with and without CGRA. All hardware parameters are equal for both systems. A significant increase of total resource consumption becomes visible. Especially the number of RAM blocks rises by a factor of 3.44. Values are also shown for all affected FUs. The amount of hardware resources consumed by the CGRA itself is relatively small. Most resources are used by the Heap FU, which contains the complete cache system. Two L1 caches with coherence controller and a L2 cache are added here for the CGRA. The resources required by the Token Machine for loop profiling and for code patch interrupt are negligible.

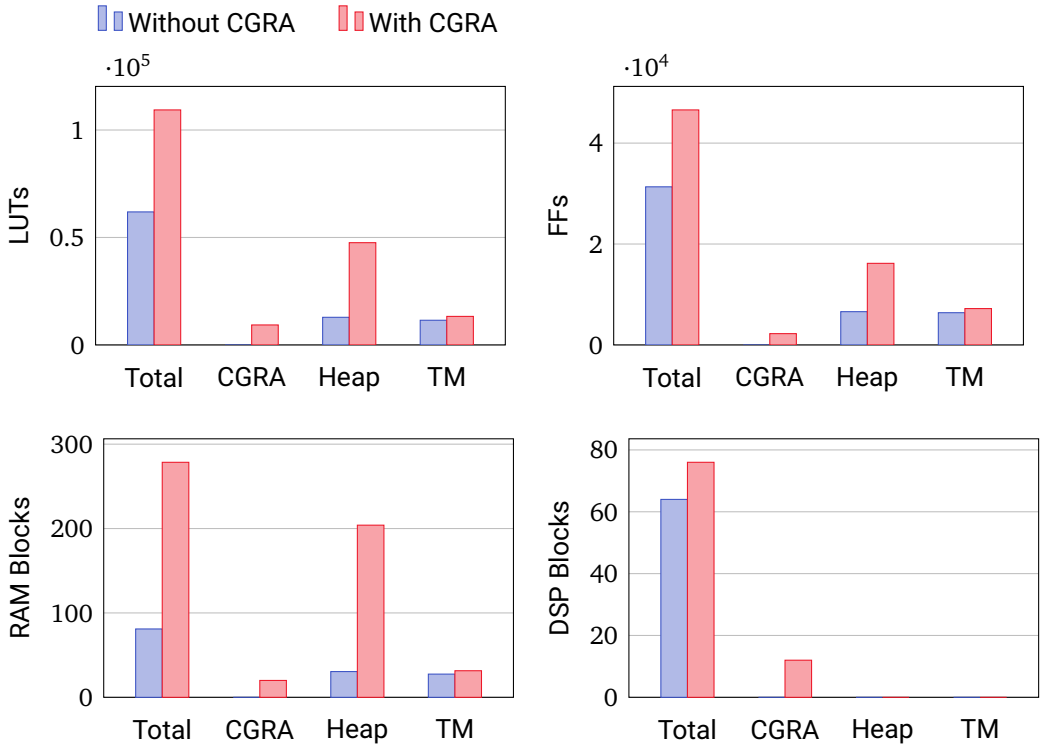


Figure 13.2.: FPGA resources consumed with and without CGRA

The processor core including CGRA is running at 50 MHz. This drop in clock frequency from 100 MHz without CGRA is not satisfying. The design does not meet timing requirements for 75 MHz, which is the next higher frequency avoiding asynchronous clock domain crossing. The critical path starts in the cache system and ends at the address input of a context memory. However, one reason for the lowered clock frequency is the high utilization of FPGA resources. 81.0 % of all available LUTs, 17.6 % of all FFs, 76.3 % of all RAM blocks, and 10.3 % of all DSP blocks are occupied. Besides optimization of the critical path, switching to a larger target FPGA will most probably improve the clock frequency considerably. In general, implementing reconfigurable hardware on top of other reconfigurable hardware is not ideal. Therefore, the ultimate goal is to realize the system using ASIC technology, which will provide further opportunities for clock frequency improvements.

14. Conclusion

This chapter gives a short summary of research contributions. Subsequently, results are discussed with regard to the objectives which have been formulated in section 4.2. Finally, opportunities for future improvements and research are highlighted.

14.1. Summary

After having analyzed weaknesses of the bytecode-based AMIDAR processor, a new ISA with name *Data Flow Oriented Java Architecture (DOJA)* has been tailored for AMIDAR. On the one hand, it is based on JVM semantics. Hence, it operates on a high abstraction level, which is beneficial for synthesizing CGRA contexts from machine code at runtime. On the other hand, it eliminates the operand stack, which has been identified as major bottleneck in executing Java bytecode. The instruction which shall receive the result of an operation is specified explicitly instead. This data flow oriented design has been chosen because the microarchitecture is data driven as well and because it avoids central memory elements for transferring data between instructions. However, DOJA is not a pure data flow architecture since instruction issuing is based on control flow.

A comprehensive software tool chain has been engineered. First, an assembly language has been defined. The corresponding assembler allows complete program images to be created. A transpiler converts Java class files into program images. Instruction scheduling has turned out to be the major challenge of this conversion process because more constraints are imposed on instruction order than with traditional architectures. Especially, preventing deadlocks requires sophisticated countermeasures. Although current algorithms cannot guarantee that valid schedules are found for all programs, they have been shown to work for many complex benchmark applications. An architecture simulator is available as well. Its timings are not accurate but it helps to verify correctness of other tool chain components.

Developing a new ISA implies implementing a corresponding processor. Many components of the bytecode-based AMIDAR processor have been retained, some key components have been redesigned. The technique for matching operands with FU operations has been changed. After having discussed several alternatives, the tag-based mechanism has been replaced by a mechanism based on small memories at FU inputs. This allows more concurrency and instruction scheduling freedom. Furthermore, a new data interconnect with double bus topology has been developed. The FUs Token Machine and Frame Stack have been re-implemented from scratch.

A fully functional FPGA prototype has been developed which can execute complex applications including CGRA context synthesis. Instruction set and hardware parameters have been optimized based on this prototype. Remarkable speedups of 1.87 for SPEC JVM98 benchmarks and 2.89 for micro benchmarks with very flat call graphs have been

measured on average. They are achieved mainly by elimination of unnecessary FU operations. Hardware resource consumption remains almost equal. The major drawback is increased code size (factor 2.82 for SPEC JVM98, 2.53 for micro benchmarks). Automatic acceleration of loops using a small CGRA with 2x2 PEs provides further speedup of 2.53 for micro benchmarks on average and 8.97 at best.

14.2. Discussion of Results

This work has been started with the question: Can Java bytecode be replaced by another ISA which exploits the AMIDAR microarchitecture better? The short answer to this question is "yes". The long answer discusses to what extent the individual research objectives have been reached.

Reduction of Unnecessary FU Operations and Data Transfers The major design goal of DOJA has been to reduce the FU operations and data transfers which serve only to pass data between instructions. The operand stack as central bottleneck has been removed by referencing target instructions for operation results explicitly. However, the Frame Stack as storage for local variables has been retained. It provides efficient replication of data. As a consequence, the total number of FU operations has been reduced by 54.0 % on average, the number of data transfers by 39.6 %. Therefore, this objective has been accomplished although not all theoretically eliminable operations and transfers have actually been eliminated.

More Concurrent Processing of Instructions Since DOJA does not have a central memory which all instructions operate on, strictly sequential execution of all instructions is not required any more. Furthermore, the new technique for matching operands with operations allows concurrent reception of operands for multiple operations. Unfortunately, concurrency is hard to measure directly. Nevertheless, analyses of exemplary instruction sequences show clear advantages. In conjunction with the reduction of FU operations and of transfers, considerable speedups of 1.87 for SPEC JVM98 benchmarks and 2.89 for micro benchmarks with flat call graphs have been achieved.

High Abstraction Level DOJA keeps the high abstraction level of Java bytecode. Semantically equivalent bytecode instructions exist for most DOJA instructions.

Opportunities for Runtime Reconfiguration Operations are assigned only to categories, not to specific FUs at compile time. Additionally, no assumptions about FU or data interconnect timing behavior are required. This enables changing the mapping of operations to FUs as well as changing FU timing at runtime. Synthesis of CGRA contexts from machine code is facilitated by the high abstraction level.

Full Support for JVM Features The new architecture provides full support for JVM features. This has been proven by running complex applications with more than 1900 classes on the processor. However, there are two weaknesses. First, hardware exceptions are not fully precise. The instruction which has caused an exception is identified precisely. But when a hardware exception is detected and handled, instructions following the

exception in program code might have already been executed. This can lead to unexpected variable values. Because hardware exceptions typically indicate software defects, which bring the program in an undefined state anyway, this is no severe limitation in practice. Second, instruction scheduling may fail for some methods although this does not happen in any of the benchmark applications. The current algorithm cannot guarantee to always find valid schedules.

Compact Instruction Encoding Converting code from Java bytecode to DOJA increases code size by a factor of 2.60 on average. Nevertheless, no bottleneck between memory and instruction decoding pipeline can be measured because of the high instruction cache hit rate. Consequently, this objective has not been reached fully but the increase in code size is considered acceptable compared to the advantages.

Moderate Hardware Requirements A slight decrease of hardware resource requirements is observed for the new architecture. Minimizing hardware resource consumption has not been a design goal. Hence, the objective has been met.

Prototype A prototype has been developed based on the same FPGA platform as the previous prototype for Java bytecode. The target clock frequency of 100 MHz has been reached. The prototype has been confirmed to execute complex applications including CGRA context synthesis.

To sum up, almost all objectives have been achieved. The first exception is the increase in code size, which is an acceptable drawback of the new architecture. The second exception is that successful conversion from Java bytecode to DOJA cannot be guaranteed under all conditions. This problem can be expected to be solved by improved instruction scheduling algorithms.

14.3. Future Work

Of course, a new processor architecture leaves plenty of room for improvements. This section points out some of the opportunities for further research and development.

14.3.1. Instruction Set Architecture

Calling Convention Passing arguments to methods could be designed more efficiently. Currently, all arguments are passed via stack memory. It might be a good idea to use direct data transfers for the first arguments instead. This would reduce unnecessary operations and data transfers further. Especially, very small methods like getters or setters would profit because they could abandon stack management.

Vector Processing Instead of transferring single data words between instructions, the basic data flow oriented principle could be extended to vectors of data words easily. This does not necessarily mean to implement full parallel processing of these vectors. Vector instructions could generate multiple tokens, one for each element. This could generate

significant speedup for applications with enough DLP without sacrificing many hardware resources.

14.3.2. Hardware Implementation

Decoding Pipeline To speed up execution when the decoding pipeline is the bottleneck (21.5 % of the time for SPEC JVM98 benchmarks, 47.8 % for micro benchmarks), a dual issue pipeline could be implemented. The high hit rate of the instruction cache shows that enough instructions are available at the beginning of the pipeline. Furthermore, many tokens can be delivered in parallel at the end of the pipeline. Hence, a dual issue pipeline could increase decoding throughput significantly. Alternatively, a token cache could be added which delivers more than one token per clock cycle. This would also increase decoding throughput by skipping the pipeline.

In order to reduce the time in which token distribution is blocked by missing instructions (9.6 % for SPEC JVM98 benchmarks, 11.7 % for micro benchmarks), branch prediction should be optimized and an invocation target cache could be introduced.

Invocations and Branches Waiting for branch and invocation parameters consumes a considerable amount of time (51.1 % for SPEC JVM98 benchmarks, 11.7 % for micro benchmarks). The exact causes of delayed Token Machine input data should be examined to find suitable solutions. Maybe, speculative execution features could be implemented to solve these problems. This could also help to support fully precise hardware exceptions.

Operand Matching and Data Interconnect Further performance enhancements could be achieved by allowing out-of-order execution on stateless FUs. However, implications to deadlock prevention must be analyzed thoroughly. Additionally, the data interconnect could support sending from one source to multiple target FUs simultaneously in order to replicate data.

CGRA Integration Currently, garbage collection fails when using an L2 data cache. There are three approaches to solve this problem. First, garbage collection could be synchronized with the L2 cache explicitly. Second, garbage collection could operate on the L2 cache instead of the external memory. Third, the L2 cache could be addressed by handles and offsets instead of physical addresses. All three approaches require a profound redesign of garbage collector or cache system. Furthermore, the critical path of designs with integrated CGRA should be improved to reach the same clock frequency with and without CGRA.

Memory Management The Heap FU should store arrays with 8 bit and 16 bit elements more efficiently. Currently, 32 bit words are allocated for these elements. This would reduce the memory consumption of some benchmark applications significantly. Positive effects on application performance can be expected because of less cache misses.

Stack memory should be assigned to threads dynamically. ISTs usually require much less stack memory than normal worker threads. Consequently, individual stack sizes would lead to more efficient stack memory usage.

14.3.3. Tool Chain

Transpiler Instruction scheduling remains the major challenge for transpilation. One problem is that each deadlock is handled individually. This strategy can fail quickly for methods with many deadlocks. A more global view on deadlocks is necessary for resolving them in such cases. Furthermore, instruction positions determined by multi-target legalization are sometimes not ideal. A better heuristic could be developed here. Moreover, the scheduler could shift problematic direct data transfers to local variable memory during preliminary scheduling. Thereby, it could be possible to implement an algorithm which can guarantee to find valid schedules for all programs.

Generic compiler optimizations could provide additional speedup. Especially, method inlining has high potential because benchmark applications with few method calls have shown better performance. Several minor improvements are possible like realizing switch statements using jump tables in constant pool. Currently, all switch statements are mapped to branch trees.

CGRA Context Synthesis Tests have shown that CGRA context synthesis is currently too slow to be applied practically at runtime. Optimizations for synthesis speed are required. Large variations in speedup by using the CGRA lead to the assumption that context synthesis can be enhanced with respect to accelerated program performance as well. Furthermore, it might be beneficial to assist runtime synthesis with additional meta information generated by the transpiler. Examples could be loop properties, handle aliasing, or potential method invocation targets.

Software Debugger A software debugger is missing among the tool chain components. The transpiler already exports all relevant information to debug files. Additionally, the Token Machine provides all necessary mechanisms. The Debugger FU should be updated and a debug agent should be developed which provides debug functionality via Java Debug Wire Protocol to connect with standard Java IDEs.

Part V.

Appendix

A. Instruction Set Listing

The following tables define all instruction operations provided by DOJA. Many of them have equivalents in Java bytecode [3]. Operations marked as “Reserved” have not been implemented yet but are planned for future improvements. “Optional” operations are not essential and can be replaced by a combination of other operations. Syntax is specified as

Port 0 Bit Width, Port 1 Bit Width, Port 2 Bit Width (Immediate Format) → Result Bit Width

with immediate formats

- u: unsigned
- s: signed
- so: signed with offset 6144

Integer Arithmetics

Mnemonic	Opcode	Syntax	Format	Category
`add`	00.0000001	32,32,— (–) → 32	R	IALU
	Adds two signed 32 bit integers.			
`add_i` (Optional)	00.1111011	32,—,— (s) → 32	R	IALU
	Adds two signed 32 bit integers. Can be replaced by `imm` *and* `add`.			
`sub`	00.0000010	32,32,— (–) → 32	R	IALU
	Subtracts two signed 32 bit integers (port 0 - port 1).			
`cmp`	00.0000011	32,32,— (–) → 32	R	IALU
	Compares two signed 32 bit integers. Port 0 > port 1 yields 1. Port 0 = port 1 yields 0. Port 0 < port 1 yields -1.			
`mul`	00.0000100	32,32,— (–) → 32	R	IALU
	Multiplies two signed 32 bit integers.			
`div`	00.0000101	32,32,— (–) → 32	R	IDIV
	Divides two signed 32 bit integers (port 0 / port 1).			
`rem`	00.0000110	32,32,— (–) → 32	R	IDIV
	Calculates the remainder of the division of two 32 bit integers (port 0 % port 1).			
`neg` (Optional)	00.0000111	32,—,— (–) → 32	R	IALU
	Multiplies a 32 bit integer by -1. Can be replaced by `imm` *and* `sub`.			

Mnemonic	Opcode	Syntax	Format	Category
`and`	00.0001000	32,32,— (–) → 32	R	IALU
	Bitwise AND operation on two 32 bit integers.			
`or`	00.0001001	32,32,— (–) → 32	R	IALU
	Bitwise OR operation on two 32 bit integers.			
`xor`	00.0001010	32,32,— (–) → 32	R	IALU
	Bitwise XOR operation on two 32 bit integers.			
`shl`	00.0001011	32,32,— (–) → 32	R	IALU
	Shifts a 32 bit integer left by a variable number of digits. (port 0 << port 1).			
`shra`	00.0001100	32,32,— (–) → 32	R	IALU
	Shifts a 32 bit integer right by a variable number of digits with sign extension (port 0 >> port 1).			
`shrl`	00.0001101	32,32,— (–) → 32	R	IALU
	Shifts a 32 bit integer right by a variable number of digits with zero extension (port 0 >>> port 1).			
`ladd`	00.0001110	64,64,— (–) → 64	R	IALU
	Adds two signed 64 bit integers.			
`lsub`	00.0001111	64,64,— (–) → 64	R	IALU
	Subtracts two signed 64 bit integers (port 0 – port 1).			
`lcmp`	00.0010000	64,64,— (–) → 32	R	IALU
	Compares two signed 64 bit integers. Port 0 > port 1 yields 1. Port 0 = port 1 yields 0. Port 0 < port 1 yields -1.			
`lmul`	00.0010001	64,64,— (–) → 64	R	IALU
	Multiplies two signed 64 bit integers.			
`ldiv`	00.0010010	64,64,— (–) → 64	R	IDIV
	Divides two signed 64 bit integers (port 0 / port 1).			
`lrem`	00.0010011	64,64,— (–) → 64	R	IDIV
	Calculates the remainder of the division of two 64 bit integers (port 0 % port 1).			
`lneg` (Optional)	00.0010100	64,—,— (–) → 64	R	IALU
	Multiplies a 64 bit integer by -1. Can be replaced by `imm`*,* `i2l`*, and* `lsub`*.*			
`land`	00.0010101	64,64,— (–) → 64	R	IALU
	Bitwise AND operation on two 64 bit integers.			
`lor`	00.0010110	64,64,— (–) → 64	R	IALU
	Bitwise OR operation on two 64 bit integers.			
`lxor`	00.0010111	64,64,— (–) → 64	R	IALU
	Bitwise XOR operation on two 64 bit integers.			

Mnemonic	Opcode	Syntax	Format	Category
`lshl`	00.0011000	64,32,— (–) → 64	R	IALU
	Shifts a 64 bit integer left by a variable number of digits (port 0 << port 1)			
`lshra`	00.0011001	64,32,— (–) → 64	R	IALU
	Shifts a 64 bit integer right by a variable number of digits with sign extension (port 0 >> port 1)			
`lshrl`	00.0011010	64,32,— (–) → 64	R	IALU
	Shifts a 64 bit integer right by a variable number of digits with zero extension (port 0 >>> port 1)			
`i2l`	00.0011011	32,—,— (–) → 64	R	IALU
	Extends a signed 32 bit integer to a 64 bit integer by sign extension.			
`l2i`	00.0011100	64,—,— (–) → 32	R	IALU
	Converts a 64 bit integer to a 32 bit integer by copying the lower 32 bits.			
`i2b`	00.0011101	32,—,— (–) → 32	R	IALU
	Copies the lower 8 bits and applies sign extension to 32 bits.			
`i2s`	00.0011110	32,—,— (–) → 32	R	IALU
	Copies the lower 16 bits and applies sign extension to 32 bits.			
`i2c`	00.0011111	32,—,— (–) → 32	R	IALU
	Copies the lower 16 bits and applies zero extension to 32 bits.			

Floating Point Arithmetics

Mnemonic	Opcode	Syntax	Format	Category
`fadd`	00.0100000	32,32,— (–) → 32	R	FPU
	Adds two single-precision floating point numbers.			
`fsub`	00.0100001	32,32,— (–) → 32	R	FPU
	Subtracts two single-precision floating point numbers (port 0 – port 1).			
`fcmp`	00.0100010	32,32,— (–) → 32	R	FPU
	Compares two single-precision floating point numbers. The result is a 32 bit integer. Port 0 > port 1 yields 1. Port 0 = port 1 yields 0. Port 0 < port 1 yields -1.			
`fmul`	00.0100011	32,32,— (–) → 32	R	FPU
	Multiplies two single-precision floating point numbers.			
`fdiv`	00.0100100	32,32,— (–) → 32	R	FDIV
	Divides two single-precision floating point numbers (port 0 / port 1).			

Mnemonic	Opcode	Syntax	Format	Category
`fneg` (Optional)	00.0100101	32,—,— (–) → 32	R	FPU
Multiplies a single-precision floating point number by -1. Can be replaced by `imm`*,* `i2f`*, and* `fsub`*.*				
`dadd`	00.0100110	64,64,— (–) → 64	R	FPU
Adds two double-precision floating point numbers.				
`dsub`	00.0100111	64,64,— (–) → 64	R	FPU
Subtracts two double-precision floating point numbers (port 0 – port 1).				
`dcmp`	00.0101000	64,64,— (–) → 32	R	FPU
Compares two double-precision floating point numbers. The result is a 32 bit integer. Port 0 > port 1 yields 1. Port 0 = port 1 yields 0. Port 0 < port 1 yields -1.				
`dmul`	00.0101001	64,64,— (–) → 64	R	FPU
Multiplies two double-precision floating point numbers.				
`ddiv`	00.0101010	64,64,— (–) → 64	R	FDIV
Divides two double-precision floating point numbers (port 0 / port 1).				
`dneg` (Optional)	00.0101011	64,—,— (–) → 64	R	FPU
Multiplies a double-precision floating point number by -1. Can be replaced by `imm`*,* `i2d`*, and* `dsub`*.*				
`f2d`	00.0101100	32,—,— (–) → 64	R	FPU
Converts a single-precision floating point number to a double-precision floating point number.				
`f2i`	00.0101101	32,—,— (–) → 32	R	FPU
Converts a single-precision floating point number to a 32 bit integer.				
`f2l`	00.0101110	32,—,— (–) → 64	R	FPU
Converts a single-precision floating point number to a 64 bit integer.				
`d2f`	00.0101111	64,—,— (–) → 32	R	FPU
Converts a double-precision floating point number to a single-precision floating point number.				
`d2i`	00.0110000	64,—,— (–) → 32	R	FPU
Converts a double-precision floating point number to a 32 bit integer.				
`d2l`	00.0110001	64,—,— (–) → 64	R	FPU
Converts a double-precision floating point number to a 64 bit integer.				

Mnemonic	Opcode	Syntax	Format	Category
`i2f`	00.0110010	32,—,— (–) → 32	R	FPU
Converts a 32 bit integer to a single-precision floating point number.				
`i2d`	00.0110011	32,—,— (–) → 64	R	FPU
Converts a 32 bit integer to a double-precision floating point number.				
`l2f`	00.0110100	64,—,— (–) → 32	R	FPU
Converts a 64 bit integer to a single-precision floating point number.				
`l2d`	00.0110101	64,—,— (–) → 64	R	FPU
Converts a 64 bit integer to a double-precision floating point number.				

Heap Memory

Mnemonic	Opcode	Syntax	Format	Category
`alloc_obj`	00.0110110	32,32,— (–) → 32	R	Heap
Allocates a new object/array on the heap. Port 0 receives the CTI. If the highest bit of the CTI is set, the array consists of 64 bit elements. Port 1 receives the size in 32 bit words. In case of arrays with 64 bit elements, the received value is multiplied by 2. The result is the handle of the new object/array.				
`getfieldh`	00.0110111	32,32,— (–) → 32	R	Heap
Gets a 32 bit field of an object. Port 0 receives the handle of the object. Port 1 receives the offset of the field in the object (number of 32 bit words). The result is flagged as handle.				
`getfieldh_i` (Optional)	00.1110100	32,—,— (u) → 32	R	Heap
Gets a 32 bit field of an object. Port 0 receives the handle of the object. The immediate value holds the offset of the field in the object (number of 32 bit words). The result is flagged as handle. This operation can be replaced by `imm` *and* `getfieldh`.				
`getfield`	00.0111000	32,32,— (–) → 32	R	Heap
Gets a 32 bit field of an object. Port 0 receives the handle of the object. Port 1 receives the offset of the field in the object (number of 32 bit words).				

Mnemonic	Opcode	Syntax	Format	Category
`getfield_i` (Optional)	00.1110101	32,—,— (u) → 32	R	Heap
Gets a 32 bit field of an object. Port 0 receives the handle of the object. The immediate value holds the offset of the field in the object (number of 32 bit words). This operation can be replaced by `imm` *and* `getfield`*.*				
`getfield64`	00.0111001	32,32,— (–) → 64	R	Heap
Gets a 64 bit field of an object. Port 0 receives the handle of the object. Port 1 receives the offset of the field in the object (number of 32 bit words).				
`getarrayh`	00.0111010	32,32,— (–) → 32	R	Heap
Gets a 32 bit element of an array. Port 0 receives the handle of the array. Port 1 receives the index of the element in the array. The result is flagged as handle.				
`getarray8` (Reserved)	00.0111011	32,32,— (–) → 32	R	Heap
Gets an 8 bit element of an array. Port 0 receives the handle of the array. Port 1 receives the index of the element in the array. The result is sign extended to 32 bits.				
`getarray16` (Reserved)	00.0111100	32,32,— (–) → 32	R	Heap
Gets a 16 bit element of an array. Port 0 receives the handle of the array. Port 1 receives the index of the element in the array. The result is sign extended to 32 bits.				
`getarray16u` (Reserved)	00.0111101	32,32,— (–) → 32	R	Heap
Gets a 16 bit element of an array. Port 0 receives the handle of the array. Port 1 receives the index of the element in the array. The result is zero extended to 32 bits.				
`getarray`	00.0111110	32,32,— (–) → 32	R	Heap
Gets a 32 bit element of an array. Port 0 receives the handle of the array. Port 1 receives the index of the element in the array.				
`getarray64`	00.0111111	32,32,— (–) → 64	R	Heap
Gets a 64 bit element of an array. Port 0 receives the handle of the array. Port 1 receives the index of the element in the array.				
`putfield`	00.1000000	32,32,32 (–)	R	Heap
Sets a 32 bit field of an object. Port 0 receives the handle of the object. Port 1 receives the offset of the field in the object (number of 32 bit words). Port 2 receives the data to write.				
`putfield_i` (Optional)	00.1110110	32,—,32 (u)	R	Heap
Sets a 32 bit field of an object. Port 0 receives the handle of the object. The immediate value holds the offset of the field in the object (number of 32 bit words). Port 2 receives the data to write. This operation can be replaced by `imm` *and* `putfield`*.*				

Mnemonic	Opcode	Syntax	Format	Category
`putfield64`	00.1000001	32,32,64 (–)	R	Heap
	Sets a 64 bit field of an object. Port 0 receives the handle of the object. Port 1 receives the offset of the field in the object (number of 32 bit words). Port 2 receives the data to write.			
`putarray8` (Reserved)	00.1000010	32,32,32 (–)	R	Heap
	Sets an 8 bit element of an array. Port 0 receives the handle of the array. Port 1 receives the index of the element in the array. Port 2 receives the data to write.			
`putarray16` (Reserved)	00.1000011	32,32,32 (–)	R	Heap
	Sets a 16 bit element of an array. Port 0 receives the handle of the array. Port 1 receives the index of the element in the array. Port 2 receives the data to write.			
`putarray`	00.1000100	32,32,32 (–)	R	Heap
	Sets a 32 bit element of an array. Port 0 receives the handle of the array. Port 1 receives the index of the element in the array. Port 2 receives the data to write.			
`putarray64`	00.1000101	32,32,64 (–)	R	Heap
	Sets a 64 bit element of an array. Port 0 receives the handle of the array. Port 1 receives the index of the element in the array. Port 2 receives the data to write.			
`getsize`	00.1000110	32,—,— (–) → 32	R	Heap
	Gets the size of an object (number of 32 bit words) or the length of an array (number of elements).			
`getcti`	00.1000111	32,—,— (–) → 32	R	Heap
	Gets the CTI of an object or an array.			
`phywrite`	00.1001000	32,—,32 (–)	R	Heap
	Writes a 32 bit word to a physical memory address. Port 0 receives the address, port 2 the data to write.			
`phyread`	00.1001001	32,—,— (–) → 32	R	Heap
	Reads a 32 bit word from a physical memory address. Port 0 receives the address.			
`lockobj`	00.1001010	32,—,— (–) → 32	R	Heap
	Prevents the object or array from being moved by garbage collection. The guaranteed physical memory address is returned as result. A second call for the same handle unlocks the object or array again.			
`flushref`	00.1001011	32,—,— (–)	R	Heap
	Flushes an object or array from cache to external memory.			
`invalidateref`	00.1001100	32,—,— (–)	R	Heap
	Invalidates the cache entries of an object or array.			

Mnemonic	Opcode	Syntax	Format	Category
`arraycopy1`	00.1001101	32,32,— (–)	R	Heap
Prepares the array copy process. Port 0 receives the handle of the source array. Port 1 receives the start offset in the source array (number of 32 bit words).				
`arraycopy2`	00.1001110	32,32,32 (–)	R	Heap
Starts the array copy process. Port 1 receives the handle of the destination array. Port 1 receives the start offset in the destination array (number of 32 bit words). Port 2 receives the number of 32 bit words to copy.				

Stack Memory

Mnemonic	Opcode	Syntax	Format	Category
`invoke_fs`	00.1011110	32,32,— (–)	R	Stack
Creates a new method frame on top of the stack and saves the return address. Port 0 receives the code address to execute after method return. Port 1 receives the size of the new frame in number of 32 bit words (excluding stack arguments, including two slots for return information).				
`invoke_fs_i` (Optional)	00.1111000	32,—,— (u)	R	Stack
Creates a new method frame on top of the stack and saves the return address. Port 0 receives the code address to execute after method return. The immediate value holds the size of the new frame in number of 32 bit words (excluding stack arguments, including two slots for return information). This operation can be replaced by `imm` *and* `invoke_fs`.				
`return_fs`	00.1011111	—,32,— (–) → 32	R	Stack
Discards the method frame at the top of the stack and restores the return address. Port 1 receives the number of 32 bit stack slots required for the arguments of the current method. The result provides the code address to execute after method return.				
`return_fs_i` (Optional)	00.1111000	—,—,— (u) → 32	R	Stack
Discards the method frame at the top of the stack and restores the return address. The immediate value holds the number of 32 bit stack slots required for the arguments of the current method. The result provides the code address to execute after method return. This operation can be replaced by `imm` *and* `return_fs`.				
`read`	00.1010001	—,—,— (s) → 32	R	Stack
Reads a 32 bit LV. The immediate value holds its address.				

Mnemonic	Opcode	Syntax	Format	Category
`read64`	00.1010010	—,—,— (s) → 64	R	Stack
	Reads a 64 bit LV. The immediate value holds its address.			
`write`	00.1010011	32,—,— (s)	R	Stack
	Writes a 32 bit LV. Port 0 receives the value to write. The immediate value holds the address.			
`write64`	00.1010100	64,—,— (s)	R	Stack
	Writes a 64 bit LV. Port 0 receives the value to write. The immediate value holds the address.			
`writer` (Optional)	00.1010101	32,—,— (s) → 32	R	Stack
	Writes a 32 bit LV and forwards it. Port 0 receives the value to write. The immediate value holds the address. The written value is returned as result. This operation can be replaced by `write` *and* `read`*.*			
`writer64` (Optional)	00.1010110	64,—,— (s) → 64	R	Stack
	Writes a 64 bit LV and forwards it. Port 0 receives the value to write. The immediate value holds the address. The written value is returned as result. This operation can be replaced by `write64` *and* `read64`*.*			
`load32`	00.1100000	—,32,— (–) → 32	R	Stack
	Reads a 32 bit LV. Port 1 receives its address.			
`load64`	00.1100001	—,32,— (–) → 64	R	Stack
	Reads a 64 bit LV. Port 1 receives its address.			
`store32`	00.1100010	32,32,— (–)	R	Stack
	Writes a 32 bit LV. Port 0 receives the value to write. Port 1 receives the address.			
`store64`	00.1100011	64,32,— (–)	R	Stack
	Writes a 64 bit LV. Port 0 receives the value to write. Port 1 receives the address.			
`push32`	00.1100100	32,—,— (–)	R	Stack
	Pushes a 32 bit value as method argument onto the stack. Port 0 receives the data to push.			
`push32_i` (Optional)	00.1110111	—,—,— (u)	R	Stack
	Pushes a constant as method argument onto the stack. The immediate value holds the constant to push. This operation can be replaced by `imm` *and* `push32`*.*			
`push64`	00.1100101	64,—,— (–)	R	Stack
	Pushes a 64 bit value as method argument onto the stack. Port 0 receives the data to push.			

Mnemonic	Opcode	Syntax	Format	Category
pop32	00.1100110	—,—,— (–) → 32	R	Stack
	Pops a 32 bit value from the stack. Currently, used for live-in transfer to the CGRA only.			
pop64	00.1100111	—,—,— (–) → 64	R	Stack
	Pops a 64 bit value from the stack. Currently, used for live-in transfer to the CGRA only.			
readpush32 (Optional)	00.1111010	—,—,— (s)	R	Stack
	Reads a 32 bit LV and pushes it as method argument onto the stack. The immediate value holds its address. This operation can be replaced by read *and* push32.			

Forwarding

Mnemonic	Opcode	Syntax	Format	Category
fwd	00.1001111	32,—,— (–) → 32	R	Forward
	Forwards a 32 bit value.			
fwd64	00.1010000	64,—,— (–) → 64	R	Forward
	Forwards a 64 bit value.			

Token Machine

Mnemonic	Opcode	Syntax	Format	Category
imm	01.0000000	—,—,— (so) → 32	I	Token Machine
	Sends the immediate value contained in the instruction.			
goto	10.0000000	—,—,— (s)	J	Token Machine
	Executes an unconditional jump. The immediate value contains the byte address of the jump target relative to the current code position.			
breq	11.101.0000	32,—,— (s)	B	Token Machine
	Executes a conditional branch if the value received on port 0 equals 0. The immediate value contains the byte address of the branch target relative to the current code position.			
brne	11.100.0000	32,—,— (s)	B	Token Machine
	Executes a conditional branch if the value received on port 0 does not equal 0. The immediate value contains the byte address of the branch target relative to the current code position.			

Mnemonic	Opcode	Syntax	Format	Category
`brg`	11.010.0000	32,—,— (s)	B	Token Machine
Executes a conditional branch if the value received on port 0 is greater than 0. The immediate value contains the byte address of the branch target relative to the current code position.				
`brl`	11.000.0000	32,—,— (s)	B	Token Machine
Executes a conditional branch if the value received on port 0 is less than 0. The immediate value contains the byte address of the branch target relative to the current code position.				
`brle`	11.001.0000	32,—,— (s)	B	Token Machine
Executes a conditional branch if the value received on port 0 is less than or equal to 0. The immediate value contains the byte address of the branch target relative to the current code position.				
`brge`	11.011.0000	32,—,— (s)	B	Token Machine
Executes a conditional branch if the value received on port 0 is greater than or equal to 0. The immediate value contains the byte address of the branch target relative to the current code position.				
`inv_static`	00.1010111	—,32,— (–) → 32	R	Token Machine
Invokes a method statically. Port 1 receives the AMTI of the method to call. The result is the address of the instruction which follows the invocation.				
`inv_static_i` (Optional)	00.1110010	—,—,— (u) → 32	R	Token Machine
Invokes a method statically. The immediate value holds the AMTI of the method to call. The result is the address of the instruction which follows the invocation. Can be replaced by `imm` *and* `inv_static`.				
`inv_virt`	00.1011000	32,32,— (–) → 32	R	Token Machine
Invokes a method virtually. Port 0 receives the CTI of the object on which the method is called. Port 1 receives the relative method table index of the method to call. The result is the address of the instruction which follows the invocation.				
`inv_virt_i` (Optional)	00.1110011	32,—,— (u) → 32	R	Token Machine
Invokes a method virtually. Port 0 receives the CTI of the object on which the method is called. The immediate value holds the relative method table index of the method to call. The result is the address of the instruction which follows the invocation. Can be replaced by `imm` *and* `inv_virt`.				

Mnemonic	Opcode	Syntax	Format	Category
`inv_interface`	00.1011001	32,32,— (–) → 32	R	Token Machine
Invokes an interface method. Port 0 receives the CTI of the object on which the method is called. Port 1 receives the relative interface method index in the lower 16 bits and the interface index in the upper 16 bits. The result is the address of the instruction which follows the invocation.				
`return_tm`	00.1011010	32,—,— (–)	R	Token Machine
Executes an unconditional jump to an absolute code address. Port 0 receives the target address. It is usually used for returning from a method.				
`const32`	00.1011011	—,32,— (–) → 32	R	Token Machine
Loads a 32 bit value from the constant pool. Port 1 receives the offset in the constant pool (in 32 bit words).				
`const64`	00.1011100	—,32,— (–) → 64	R	Token Machine
Loads a 64 bit value from the constant pool. Port 1 receives the offset in the constant pool (in 32 bit words).				
`instanceof`	00.1011101	32,32,— (–) → 32	R	Token Machine
Checks if a class is derived from another class or implements an interface. Port 0 receives the CTI of the child class. Port 1 receives the CTI of the parent class. The result is 0 if the check has failed. Otherwise, the result is unequal to 0.				

Thread Scheduler

Mnemonic	Opcode	Syntax	Format	Category
`inv_scheduler`	00.1101000	32,32,64 (–) → 32	R	Thread Scheduler
Executes a non-blocking scheduler operation. Port 0 receives the scheduler specific operation code. Port 1 and 2 receive parameters which depend on the operation code. The result also depends on the operation code [15].				
`binv_scheduler`	00.1101001	32,32,64 (–)	R	Thread Scheduler
Executes a blocking scheduler operation. Port 0 receives the scheduler specific operation code. Port 1 and 2 receive parameters which depend on the operation code [15].				
`monitorenter`	00.1101010	32,—,— (–)	R	Thread Scheduler
Acquires the monitor which belongs to the handle on port 0. The current thread is suspended as long as the monitor cannot be acquired.				
`monitorexit`	00.1101011	32,—,— (–)	R	Thread Scheduler
Releases the monitor which belongs to the handle on port 0.				

CGRA

Mnemonic	Opcode	Syntax	Format	Category
`cgra_init`	00.1101100	32,—,— (–)	R	CGRA
Initializes a CGRA kernel. Port 0 receives the kernel ID. Must be executed before live-in values can be pushed.				
`cgra_run`	00.1101101	32,—,— (–) → 32	R	CGRA
Starts kernel execution on the CGRA. Port 0 receives an arbitrary value to synchronize the CGRA to normal program execution. 0 is sent as result after kernel execution has been finished. This dummy result allows to synchronize normal program execution to the CGRA.				
`cgra_push32`	00.1101110	32,—,— (–)	R	CGRA
Transfers a 32 bit live-in value to the CGRA.				
`cgra_push64`	00.1101111	64,—,— (–)	R	CGRA
Transfers a 64 bit live-in value to the CGRA.				
`cgra_pull32`	00.1110000	—,—,— (–) → 32	R	CGRA
Transfers a 32 bit live-out value from the CGRA.				
`cgra_pull64`	00.1110001	—,—,— (–) → 64	R	CGRA
Transfers a 64 bit live-out value from the CGRA.				
`cgra_pullh`	00.1111100	—,—,— (–) → 32	R	CGRA
Transfers a 32 bit live-out value from the CGRA. The live-out value is flagged as handle.				

Special

Mnemonic	Opcode	Syntax	Format	Category
`nop`	00.0000000	—,—,— (–)	R	undefined
Does not execute any operation and discards all received values. An arbitrary number of values can be received. Their target ports are irrelevant.				
`disc`	00.0000000	32,—,— (–)	R	undefined
Same behavior as `nop`*. The syntactic difference to* `nop` *is relevant for static code checks only.*				
`disc64`	00.0000000	64,—,— (–)	R	undefined
Same behavior as `nop`*. The syntactic difference to* `nop` *is relevant for static code checks only.*				

Mnemonic	Opcode	Syntax	Format	Category
`send_again`	00.1111101	—,—,— (–) → 32	R	undefined
	Sends a 32 bit value which has been duplicated by a keep result flag.			
`send_again64`	00.1111110	—,—,— (–) → 64	R	undefined
	Sends a 64 bit value which has been duplicated by a keep result flag.			
`breakpoint`	00.1111111	—,—,— (–)	R	undefined
	Stops decoding before this instruction and sends a signal to the debugger.			

B. Constant Hardware Parameters

The DOJA processor has some parameters which are not discussed in this work and are treated as constants. This chapter lists their values. Cache and heap parameters have been inherited from the bytecode processor. The fetch queue has been sized to store one instruction cache line. Reasonable sizes which do not impact performance significantly have been determined for the decode and immediate queues experimentally.

Cache	Sets	Ways	Words per Line	Size
Heap Data L1 (Main)	512	4	8	64 KiB
Heap Data L1 (CGRA)	128	4	8	16 KiB
Heap Data L2	2048	4	8	256 KiB
Handle Table	512	2	3	12 KiB
Instructions	256	4	16	64 KiB
Constant Pool	512	1	1	2 KiB
Class Table	512	1	4	8 KiB
Interface Table	512	1	1	2 KiB
Implemented Interfaces Table	512	1	1	2 KiB
Method Table	512	1	1	2 KiB

Table B.1.: Cache configurations

Parameter	Value
Heap Data Size	450 MB
Number of Handles	$2^{22} = 4\,194\,304$

Table B.2.: Heap configuration

Queue	Entries
Fetch Queue	16
Decode Queue	4
Immediate Queue	8

Table B.3.: Queue sizes

C. References

[1] Apple Inc. *Apple Unleashes M1*. Nov. 10, 2020. URL: https://www.apple.com/newsroom/2020/11/apple-unleashes-m1/ (visited on 05/21/2021).

[2] Dennis Leander Wolf et al. "AMIDAR Project: Lessons Learned in 15 Years of Researching Adaptive Processors". In: *2018 13th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*. July 2018, pp. 1–8. DOI: 10.1109/ReCoSoC.2018.8449384.

[3] Tim Lindholm et al. *The Java Virtual Machine Specification, Java SE 8 Edition*. Feb. 13, 2015. URL: https://docs.oracle.com/javase/specs/jvms/se8/html/index.html (visited on 11/08/2020).

[4] OpenJDK Project. *The Java HotSpot Virtual Machine*. URL: http://openjdk.java.net/groups/hotspot/ (visited on 05/22/2021).

[5] Stephan Gatzka and Christian Hochberger. "The AMIDAR Class of Reconfigurable Processors". In: *The Journal of Supercomputing* 32.2 (May 1, 2005), pp. 163–181. ISSN: 1573-0484. DOI: 10.1007/s11227-005-0290-3.

[6] J.M. O'Connor and M. Tremblay. "picoJava-I: The Java Virtual Machine in Hardware". In: *IEEE Micro* 17.2 (Mar. 1997), pp. 45–53. ISSN: 1937-4143. DOI: 10.1109/40.592314.

[7] Flavius Gruian and Mark Westmijze. "Investigating Hardware Micro-Instruction Folding in a Java Embedded Processor". In: *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*. JTRES '10. New York, NY, USA: Association for Computing Machinery, Aug. 19, 2010, pp. 102–108. ISBN: 978-1-4503-0122-0. DOI: 10.1145/1850771.1850787.

[8] Alexander Schwarz and Christian Hochberger. "Engineering an Optimized Instruction Set Architecture for AMIDAR Processors". In: *Architecture of Computing Systems – ARCS 2020*. Ed. by André Brinkmann et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 124–137. ISBN: 978-3-030-52794-5. DOI: 10.1007/978-3-030-52794-5_10.

[9] Alexander Schwarz and Christian Hochberger. "Performance Gain of a Data Flow Oriented ISA as Replacement for Java Bytecode". In: *Architecture of Computing Systems – ARCS 2021*. Ed. by Christian Hochberger, Lars Bauer, and Thilo Pionteck. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2021, pp. 103–117. ISBN: 978-3-030-81682-7. DOI: 10.1007/978-3-030-81682-7_7.

[10] Oracle. *Java Native Interface Specification*. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/jniTOC.html (visited on 05/26/2021).

[11] Oracle. *JSR 360: Connected Limited Device Configuration 8*. Apr. 30, 2014. URL: https://jcp.org/en/jsr/detail?id=360 (visited on 05/26/2021).

[12] Oracle. *JSR 361: Java ME Embedded Profile*. Apr. 30, 2014. URL: https://jcp.org/en/jsr/detail?id=361 (visited on 05/26/2021).

[13] James Gosling et al. *The Java Language Specification, Java SE 8 Edition*. Feb. 13, 2015. URL: https://docs.oracle.com/javase/specs/jls/se8/html/index.html (visited on 05/26/2021).

[14] Android Open Source Project. *Android Runtime (ART) and Dalvik*. URL: https://source.android.com/devices/tech/dalvik (visited on 11/08/2020).

[15] Changgong Li. "Implementation of an AMIDAR-based Java Processor". Darmstadt: Technische Universität Darmstadt, 2019. URL: https://tuprints.ulb.tu-darmstadt.de/8627.

[16] Lukas Johannes Jung. "Optimization of the Memory Subsystem of a Coarse Grained Reconfigurable Hardware Accelerator". Darmstadt: Technische Universität Darmstadt, 2019. URL: https://tuprints.ulb.tu-darmstadt.de/8674.

[17] Changgong Li, Alexander Schwarz, and Christian Hochberger. "A Readback Based General Debugging Framework for Soft-Core Processors". In: *2016 IEEE 34th International Conference on Computer Design (ICCD)*. Oct. 2016, pp. 568–575. DOI: 10.1109/ICCD.2016.7753342.

[18] ARM Limited. *AMBA AXI and ACE Protocol Specification, Issue H*. Technical Specification. Mar. 31, 2020. URL: https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/IHI0022H_amba_axi_protocol_spec.pdf (visited on 05/28/2021).

[19] Richard Herveille. *WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*. Technical Specification. Sept. 7, 2002. URL: http://cdn.opencores.org/downloads/wbspec_b3.pdf (visited on 05/28/2021).

[20] Esko Luontola. *Retrolambda*. Version 2.5.6. Nov. 30, 2018. URL: https://github.com/luontola/retrolambda (visited on 05/27/2021).

[21] Ramon Wirsch and Christian Hochberger. "Towards Transparent Dynamic Binary Translation from RISC-V to a CGRA". In: *Architecture of Computing Systems - ARCS2021*. Ed. by Christian Hochberger, Lars Bauer, and Thilo Pionteck. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2021, pp. 118–132. ISBN: 978-3-030-81682-7. DOI: 10.1007/978-3-030-81682-7_8.

[22] James Archibald and Jean-Loup Baer. "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model". In: *ACM Transactions on Computer Systems* 4.4 (Sept. 1, 1986), pp. 273–298. ISSN: 0734-2071. DOI: 10.1145/6513.6514.

[23] ARM Limited. *Jazelle - ARM Architecture Extensions for Java Applications*. Whitepaper.

[24] I. Sideris, K. Pekmestzi, and G. Economakos. "An Instruction Set Extension for Java Bytecodes Translation Acceleration". In: *2008 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*. July 2008, pp. 116–123. DOI: 10.1109/ICSAMOS.2008.4664854.

[25] Jong-Sung Lee and H. Kim. “The Javelin: A Java Accelerator Based on Hardware Translation Method”. In: *2008 International SoC Design Conference*. Vol. 01. Nov. 2008, pp. I-366-I–369. DOI: 10.1109/SOCDC.2008.4815648.

[26] Harlan McGhan and Mike O’Connor. “PicoJava: A Direct Execution Engine For Java Bytecode”. In: *Computer* 31.10 (Oct. 1, 1998), pp. 22–30. ISSN: 0018-9162. DOI: 10.1109/2.722273.

[27] Sun Microsystems. *picoJava-II Microarchitecture Guide*. 1999.

[28] Sun Microsystems. *picoJava-II Programmer’s Reference Manual*. 1999.

[29] J. Kreuzinger et al. “Real-Time Event-Handling and Scheduling on a Multithreaded Java Microcontroller”. In: *Microprocessors and Microsystems* 27.1 (Feb. 1, 2003), pp. 19–31. ISSN: 0141-9331. DOI: 10.1016/S0141-9331(02)00082-0.

[30] Sascha Uhrig and Jörg Wiese. “Jamuth: An IP Processor Core for Embedded Java Real-Time Systems”. In: *Proceedings of the 5th International Workshop on Java Technologies for Real-Time and Embedded Systems*. JTRES ’07. New York, NY, USA: Association for Computing Machinery, Sept. 26, 2007, pp. 230–237. ISBN: 978-1-59593-813-8. DOI: 10.1145/1288940.1288974.

[31] Martin Schoeberl. “A Java Processor Architecture for Embedded Real-Time Systems”. In: *Journal of Systems Architecture* 54.1 (Jan. 1, 2008), pp. 265–286. ISSN: 1383-7621. DOI: 10.1016/j.sysarc.2007.06.001.

[32] Martin Schoeberl. *JOP Reference Handbook*. CreateSpace, 2009. URL: https://www.jopdesign.com/doc/handbook.pdf (visited on 11/08/2020).

[33] Thomas B. Preußer, Martin Zabel, and P. Reichel. *The SHAP Microarchitecture and Java Virtual Machine*. Technical Report. 2007.

[34] aJile Systems. *Real-Time, Low Power Network, Multimedia Direct Execution Microprocessor For The JME Platform aJ-200*. Technical Reference Manual. 2010. URL: https://www.ajile.com/downloads/aJ-200V1.pdf (visited on 11/08/2020).

[35] Chun-Jen Tsai et al. “A Java Processor IP Design for Embedded SoC”. In: *ACM Transactions on Embedded Computing Systems* 14.2 (Feb. 17, 2015), 35:1–35:25. ISSN: 1539-9087. DOI: 10.1145/2629649.

[36] C. Tsai et al. “Hardwiring the OS Kernel into a Java Application Processor”. In: *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. July 2017, pp. 53–60. DOI: 10.1109/ASAP.2017.7995259.

[37] Martin Schoeberl, Thomas B. Preußer, and Sascha Uhrig. “The Embedded Java Benchmark Suite JemBench”. In: *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*. JTRES ’10. New York, NY, USA: Association for Computing Machinery, Aug. 19, 2010, pp. 120–127. ISBN: 978-1-4503-0122-0. DOI: 10.1145/1850771.1850789.

[38] Wolfgang Puffitsch and Martin Schoeberl. “picoJava-II in an FPGA”. In: *Proceedings of the 5th International Workshop on Java Technologies for Real-Time and Embedded Systems*. JTRES ’07. New York, NY, USA: Association for Computing Machinery, Sept. 26, 2007, pp. 213–221. ISBN: 978-1-59593-813-8. DOI: 10.1145/1288940.1288972.

[39] Standard Performance Evaluation Corporation. *SPEC JVM98 Benchmarks*. URL: http://www.spec.org/jvm98/ (visited on 05/09/2021).

[40] Lee-Ren Ton et al. "Design of an Optimal Folding Mechanism for Java Processors". In: *Microprocessors and Microsystems* 26.8 (Nov. 10, 2002), pp. 341–352. ISSN: 0141-9331. DOI: 10.1016/S0141-9331(02)00042-X.

[41] L.-C. Chang et al. "Stack Operations Folding in Java Processors". In: *IEE Proceedings - Computers and Digital Techniques* 145.5 (Sept. 1, 1998), pp. 333–340. ISSN: 1359-7027. DOI: 10.1049/ip-cdt:19982200.

[42] Lee-Ren Ton et al. "A Software/Hardware Cooperated Stack Operations Folding Model for Java Processors". In: *Journal of Systems and Software* 72.3 (Aug. 1, 2004), pp. 377–387. ISSN: 0164-1212. DOI: 10.1016/S0164-1212(03)00100-6.

[43] A. Kim and M. Chang. "Advanced POC Model-Based Java Instruction Folding Mechanism". In: *Proceedings of the 26th Euromicro Conference. EUROMICRO 2000. Informatics: Inventing the Future*. Vol. 1. Sept. 2000, 332–338 vol.1. DOI: 10.1109/EURMIC.2000.874650.

[44] Ramesh Radhakrishnan, Deependra Talla, and Lizy Kurian John. "Allowing for ILP in an Embedded Java Processor". In: *Proceedings of the 27th Annual International Symposium on Computer Architecture*. ISCA '00. New York, NY, USA: Association for Computing Machinery, May 1, 2000, pp. 294–305. ISBN: 978-1-58113-232-8. DOI: 10.1145/339647.339702.

[45] C. Hochberger et al. "Synthilation: JIT-compilation of Microinstruction Sequences in AMIDAR Processors". In: *Proceedings of the 2014 Conference on Design and Architectures for Signal and Image Processing*. Oct. 2014, pp. 1–6. DOI: 10.1109/DASIP.2014.7115634.

[46] Richard M. Karp and Rayamond E. Miller. "Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing". In: *SIAM Journal on Applied Mathematics* 14.6 (Nov. 1, 1966), pp. 1390–1411. ISSN: 0036-1399. DOI: 10.1137/0114108.

[47] Duane Albert Adams. "A Computation Model with Data Flow Sequencing". PhD thesis. Stanford, CA, USA: Stanford University, 1969.

[48] J. E. Rodrigues. *A Graph Model for Parallel Computations*. Technical Report. USA: Massachusetts Institute of Technology, 1969.

[49] Jack B. Dennis. "First Version of a Data Flow Procedure Language". In: *Programming Symposium*. Ed. by B. Robinet. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1974, pp. 362–376. ISBN: 978-3-540-37819-8. DOI: 10.1007/3-540-06859-7_145.

[50] Steven Swanson et al. "The WaveScalar Architecture". In: *ACM Transactions on Computer Systems* 25.2 (May 1, 2007), 4:1–4:54. ISSN: 0734-2071. DOI: 10.1145/1233307.1233308.

[51] Jack B. Dennis and David P. Misunas. "A Preliminary Architecture for a Basic Data-Flow Processor". In: *ACM SIGARCH Computer Architecture News* 3.4 (Dec. 1, 1974), pp. 126–132. ISSN: 0163-5964. DOI: 10.1145/641675.642111.

[52] Jack B. Dennis. "Data Flow Supercomputers". In: *Computer* 13.11 (Nov. 1980), pp. 48–56. ISSN: 1558-0814. DOI: 10.1109/MC.1980.1653418.

[53] W. B. Ackermann. “A Structure Processing Facility for Dataflow Computers”. In: *Proc. of the International Conference on Parallel Processing*. Aug. 1978, pp. 166–172.

[54] J. R Gurd, C. C. Kirkham, and I. Watson. “The Manchester Prototype Dataflow Computer”. In: *Communications of the ACM* 28.1 (Jan. 2, 1985), pp. 34–52. ISSN: 0001-0782. DOI: 10.1145/2465.2468.

[55] L. M. Patnaik, R. Govindarajan, and N. S. Ramadoss. “Design and Performance Evaluation of EXMAN: An EXtended MANchester Data Flow Computer”. In: *IEEE Transactions on Computers* C-35.3 (Mar. 1986), pp. 229–244. ISSN: 1557-9956. DOI: 10.1109/TC.1986.1676747.

[56] G. M. Papadopoulos and D. E. Culler. “Monsoon: An Explicit Token-Store Architecture”. In: *Proceedings of the 17th Annual International Symposium on Computer Architecture*. May 1990, pp. 82–91. DOI: 10.1109/ISCA.1990.134511.

[57] Arvind and R. S. Nikhil. “Executing a Program on the MIT Tagged-Token Dataflow Architecture”. In: *IEEE Transactions on Computers* 39.3 (Mar. 1990), pp. 300–318. ISSN: 1557-9956. DOI: 10.1109/12.48862.

[58] V. G. Grafe and J. E. Hoch. “The Epsilon-2 Hybrid Dataflow Architecture”. In: *Digest of Papers Compcon Spring ’90. Thirty-Fifth IEEE Computer Society International Conference on Intellectual Leverage*. Feb. 1990, pp. 88–93. DOI: 10.1109/CMPCON.1990.63658.

[59] D. Burger et al. “Scaling to the End of Silicon with EDGE Architectures”. In: *Computer* 37.7 (July 2004), pp. 44–55. ISSN: 1558-0814. DOI: 10.1109/MC.2004.65.

[60] Karthikeyan Sankaralingam et al. “TRIPS: A Polymorphous Architecture for Exploiting ILP, TLP, and DLP”. In: *ACM Transactions on Architecture and Code Optimization* 1.1 (Mar. 1, 2004), pp. 62–93. ISSN: 1544-3566. DOI: 10.1145/980152.980156.

[61] Mark Gebhart et al. “An Evaluation of the TRIPS Computer System”. In: *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XIV. New York, NY, USA: Association for Computing Machinery, Mar. 7, 2009, pp. 1–12. ISBN: 978-1-60558-406-5. DOI: 10.1145/1508244.1508246.

[62] R. M. Tomasulo. “An Efficient Algorithm for Exploiting Multiple Arithmetic Units”. In: *IBM Journal of Research and Development* 11.1 (Jan. 1967), pp. 25–33. ISSN: 0018-8646. DOI: 10.1147/rd.111.0025.

[63] Y. N. Patt, W. M. Hwu, and M. Shebanow. “HPS, a New Microarchitecture: Rationale and Introduction”. In: *ACM SIGMICRO Newsletter* 16.4 (Dec. 1, 1985), pp. 103–108. ISSN: 1050-916X. DOI: 10.1145/18906.18916.

[64] F. Yazdanpanah et al. “Hybrid Dataflow/von-Neumann Architectures”. In: *IEEE Transactions on Parallel and Distributed Systems* 25.6 (June 2014), pp. 1489–1509. ISSN: 1558-2183. DOI: 10.1109/TPDS.2013.125.

[65] D. D. Gajski et al. “A Second Opinion on Data Flow Machines and Languages”. In: *Computer* 15.2 (Feb. 1982), pp. 58–69. ISSN: 1558-0814. DOI: 10.1109/MC.1982.1653942.

[66] Ben Lee and A. R. Hurson. “Issues in Dataflow Computing”. In: *Advances in Computers*. Ed. by Marshall C. Yovits. Vol. 37. Elsevier, Jan. 1, 1993, pp. 285–333. DOI: 10.1016/S0065-2458(08)60407-6.

[67] Henk Corporaal. *Microprocessor Architectures: From VLIW to TTA*. USA: John Wiley & Sons, Inc., 1997. 428 pp. ISBN: 978-0-471-97157-3.

[68] Pekka Jääskeläinen et al. “Transport-Triggered Soft Cores”. In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. May 2018, pp. 83–90. DOI: 10.1109/IPDPSW.2018.00022.

[69] D. Tabak and G. J. Lipovski. “MOVE Architecture in Digital Controllers”. In: *IEEE Journal of Solid-State Circuits* 15.1 (Feb. 1980), pp. 116–126. ISSN: 1558-173X. DOI: 10.1109/JSSC.1980.1051344.

[70] Otto Esko et al. “Customized Exposed Datapath Soft-Core Design Flow with Compiler Support”. In: *2010 International Conference on Field Programmable Logic and Applications*. Aug. 2010, pp. 217–222. DOI: 10.1109/FPL.2010.51.

[71] J. Heikkinen et al. “Dictionary-Based Program Compression on Transport Triggered Architectures”. In: *2005 IEEE International Symposium on Circuits and Systems*. May 2005, 1122–1125 Vol. 2. DOI: 10.1109/ISCAS.2005.1464790.

[72] J. Wei et al. “Program Compression Based on Arithmetic Coding on Transport Triggered Architecture”. In: *2008 International Conference on Embedded Software and Systems Symposia*. July 2008, pp. 126–131. DOI: 10.1109/ICESS.Symposia.2008.9.

[73] Janne Helkala et al. “Variable Length Instruction Compression on Transport Triggered Architectures”. In: *2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. July 2014, pp. 149–155. DOI: 10.1109/SAMOS.2014.6893206.

[74] Yifan He et al. “MOVE-Pro: A Low Power and High Code Density TTA Architecture”. In: *2011 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. July 2011, pp. 294–301. DOI: 10.1109/SAMOS.2011.6045474.

[75] Heikki O. Kultala et al. “Exposed Datapath Optimizations for Loop Scheduling”. In: *2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. July 2017, pp. 171–178. DOI: 10.1109/SAMOS.2017.8344625.

[76] Maxim Integrated. *MAXQ Family User’s Guide*. URL: https://pdfserv.maximintegrated.com/en/an/AN4811.pdf (visited on 04/30/2021).

[77] Philipp Müller. *Untersuchung verschiedener Daten-Synchronisationsmechanismen für AMIDAR-Prozessoren*. Bachelor Thesis. Technische Universität Darmstadt, Feb. 14, 2019.

[78] S. Gatzka and C. Hochberger. “Hardware Based Online Profiling in AMIDAR Processors”. In: *19th IEEE International Parallel and Distributed Processing Symposium*. Apr. 2005. DOI: 10.1109/IPDPS.2005.239.

[79] Oren Ben-Kiki, Clark Evans, and Ingy döt Net. *YAML Specification 1.2.1*. Oct. 1, 2009. URL: https://yaml.org/spec/1.2.1/ (visited on 09/23/2021).

[80] Troy Downing and Jon Meyer. *Java Virtual Machine*. 1. Edition. Cambridge, Mass.: O'Reilly Media, Apr. 11, 1997. 450 pp. ISBN: 978-1-56592-194-8.

[81] Jonathan Meyer, Daniel Reynaud, and Iouri Kharon. *Jasmin*. 2004. URL: http://jasmin.sourceforge.net/ (visited on 05/11/2021).

[82] *JavaCC*. Version 6.1.2. May 8, 2014. URL: https://javacc.github.io/javacc/ (visited on 07/05/2021).

[83] Patrick Lam et al. "The Soot Framework for Java Program Analysis: A Retrospective". In: Cetus Users and Compiler Infastructure Workshop (CETUS 2011). Oct. 2011.

[84] *Soot*. Version 4.2.1. Oct. 28, 2020. URL: https://github.com/soot-oss/soot (visited on 07/07/2021).

[85] Preston Briggs et al. "Practical Improvements to the Construction and Destruction of Static Single Assignment Form". In: *Software: Practice and Experience* 28.8 (1998), pp. 859–881. ISSN: 1097-024X. DOI: 10.1002/(SICI)1097-024X(19980710)28:8<859::AID-SPE188>3.0.CO;2-8.

[86] Jakob Stoklund Olesen. *Greedy Register Allocation in LLVM 3.0*. Sept. 18, 2011. URL: https://blog.llvm.org/2011/09/greedy-register-allocation-in-llvm-30.html (visited on 07/19/2021).

[87] Digilent. *Nexys Video Reference Manual*. URL: https://digilent.com/reference/programmable-logic/nexys-video/reference-manual (visited on 08/31/2021).

D. Own Publications

Changgong Li, Alexander Schwarz, and Christian Hochberger. "A Readback Based General Debugging Framework for Soft-Core Processors". In: *2016 IEEE 34th International Conference on Computer Design (ICCD)*. Oct. 2016, pp. 568–575. DOI: 10.1109/ICCD.2016.7753342.

Alexander Schwarz and Christian Hochberger. "Engineering an Optimized Instruction Set Architecture for AMIDAR Processors". In: *Architecture of Computing Systems – ARCS 2020*. Ed. by André Brinkmann et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 124–137. ISBN: 978-3-030-52794-5. DOI: 10.1007/978-3-030-52794-5_10.

Alexander Schwarz and Christian Hochberger. "Performance Gain of a Data Flow Oriented ISA as Replacement for Java Bytecode". In: *Architecture of Computing Systems – ARCS 2021*. Ed. by Christian Hochberger, Lars Bauer, and Thilo Pionteck. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2021, pp. 103–117. ISBN: 978-3-030-81682-7. DOI: 10.1007/978-3-030-81682-7_7.

E. Supervised Theses

Ramon Wirsch. “Implementation of a Simulator for AMIDAR Under Application of a New Instruction Set Architecture”. Master Thesis. Technische Universität Darmstadt, Oct. 2017.

Philipp Müller. “Statische Analyse der Datenkonflikte bei AMIDAR-Prozessoren mit neuer ISA”. Project Seminar. Technische Universität Darmstadt, June 2018.

Philipp Müller. “Untersuchung verschiedener Daten-Synchronisationsmechanismen für AMIDAR-Prozessoren”. Bachelor Thesis. Technische Universität Darmstadt, Feb. 2019.

Hendrik Schöffmann. “Implementierung eines Graphgenerators für die AMIDAR Instruction Set Architecture”. Master Thesis. Technische Universität Darmstadt, Mar. 2019.

Yannik Böttcher. “Untersuchung verschiedener Multithreading-Mechanismen für die neue AMIDAR ISA”. Project Seminar. Technische Universität Darmstadt, May 2019.

Felix Retsch. “Entwicklung einer binären Codierung für die AMIDAR Instruction Set Architecture”. Project Seminar. Technische Universität Darmstadt, Aug. 2019.

Adrian Weber. “Implementierung eines Instruction Schedulers für die AMIDAR Instruction Set Architecture”. Master Thesis. Technische Universität Darmstadt, Sep. 2019.

Yannik Böttcher. “Integration von Multithreading in den Simulator für AMIDAR-Prozessoren”. Bachelor Thesis. Technische Universität Darmstadt, Oct. 2019.

Felix Retsch. “Entwicklung eines Token Generators für die neue AMIDAR Instruction Set Architecture”. Master Thesis. Technische Universität Darmstadt, Mar. 2020.

Maximilian Heer. “Entwicklung zweier Speichermodule für die AMIDAR New Instruction Set Architecture”. Project Seminar. Technische Universität Darmstadt, Apr. 2020.

Maximilian Heer. “Integration von Multithreading in den AMIDAR-Prozessor mit neuer ISA”. Bachelor Thesis. Technische Universität Darmstadt, Oct. 2020.

Sebastian Kleemann. “Kompakteres Speicherlayout für den Heap in AMIDAR Prozessoren”. Project Seminar. Technische Universität Darmstadt, Mar. 2021.

F. List of Abbreviations

AMIDAR	Adaptive Microinstruction Driven Architecture
AMTI	Absolute Method Table Index
AXT	AMIDAR Executable
BRAM	RAM Block
CDFG	Control and Data Flow Graph
CFG	Control Flow Graph
CGRA	Coarse Grained Reconfigurable Array
CTI	Class Table Index
DAG	Directed Acyclic Graph
DBG	Debugger
DLP	Data Level Parallelism
DMA	Direct Memory Access
DOJA	Data Flow Oriented Java Architecture
EBNF	Extended Backus-Naur Form
FDIV	Floating Point Divider
FF	Flip-Flop
FP	Frame Pointer
FPU	Floating Point Unit
FS	Frame Stack
FSM	Finite State Machine
FU	Functional Unit
FWD	Forward Unit
IALU	Integer ALU
IDIV	Integer Divider
ILP	Instruction Level Parallelism
IMUL	Integer Multiplier
ISA	Instruction Set Architecture
IST	Interrupt Service Thread
JAR	Java Archive
JIT	Just-In-Time Compilation
JVM	Java Virtual Machine
LALU	Long ALU
LCU	Local Configuration Unit
LUT	Lookup Table
LV	Local Variable
MMTI	Method Meta Table Index
NAX	New AMIDAR Executable

PE	Processing Element
SCAR	Scheduler Application Representation
SM	Structure Memory
SP	Stack Pointer
SPM	Scratch Pad Memory
SSA	Static Single Assignment
TDN	Token Distribution Network
TLP	Thread Level Parallelism
TM	Token Machine
TS	Thread Scheduler
TTA	Transport Triggered Architecture
VLIW	Very Large Instruction Word