Energy Efficient Code Generation for Streaming Applications

PROEFSCHRIFT

der verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de rector magnificus, prof.dr.ir. C.J. van Duijn, voor een commissie aangewezen door het College voor Promoties in het openbaar te verdedigen op donderdag 27 november 2014 om 16.00 uur

door

Dongrui She

geboren te Guangdong, China
Dit proefschrift is goedgekeurd door de promotoren en de samenstelling van de promotiecommissie is als volgt:

voorzitter: prof.dr.ir. A.C.P.M. Backx
1e promotor: prof.dr. H. Corporaal
copromotor: dr. M. Beemster (ACE Associated Compiler Experts bv)
leden: dr. C. Silvano (Politecnico di Milano)
        prof.dr.ir. H.J. Sips (TUD)
        prof.dr. K.G.W. Goossens
        prof.dr.ir. C.H. van Berkel
adviseur: dr. A. Terechko (Vector Fabrics B.V.)
Energy Efficient Code Generation for Streaming Applications
This work is supported in part by the Dutch Technology Foundation STW, project NEST 10346.

© Copyright 2014, Dongrui She
All rights reserved. Reproduction in whole or in part is prohibited without the written consent of the copyright owner.

Printing: Ridderprint BV
A catalogue record is available from the Eindhoven University of Technology Library.
ISBN: 978-90-386-3733-4
NUR: 959
Abstract

Contemporary mobile devices such as a smart-phone often need to run multiple applications with high performance demand, for example, wireless communication and high-definition video codecs. However, such devices often run on limited power sources like batteries. As a result, energy efficiency has always been important in embedded system design. To achieve high efficiency in these systems, significant efforts at different levels are required. As a lot of applications on embedded systems fall into the category of streaming application, which perform the same or similar operations on regular sequences of data. Therefore optimizing for streaming applications is of great importance. In this thesis, we propose processor architecture and code generation techniques to tackle part of the challenges in designing energy efficient processing systems for streaming applications.

Firstly, we attempt to reduce the energy consumption of the register file (RF), which is typically one of the most power-hungry components in a processor. Analysis reveals that in many applications, most variables are used locally only for a few times, resulting in a lot of RF accesses that can be eliminated. In this work, we introduce an explicit datapath architecture that allows software to directly control the bypassing network. As the software has fine-grained control over the datapath, efficient code generation is a key to achieve high energy efficiency. We propose compiler back-end for the proposed explicit bypassing architecture. The compiler includes algorithms to schedule instructions such that most of the unnecessary RF accesses are eliminated, while the performance is unaffected. Experimental results show that the total energy consumption is reduced by 9.19% compared to the RISC baseline, and is more efficient than TTA-based processors with similar amount of resources.

Secondly, a method to support flexible operation-pair pattern, which consists of two simple operations, in a RISC-like processor with compact 24-bit instruction set architecture (ISA) is presented. Two problems are tackled: i) encoding large number of special operation opcodes; ii) supplying sufficient data to the special function unit (SFU). Application analysis shows that operation-pair patterns have good locality in many applications. Therefore we propose a partially re-configurable instruction decoder that supports flexible operation pairs with only ten opcodes. Explicit bypassing is used to reduce the overhead of supplying more operands to the SFU. The efficiency of proposed solution relies on the compiler. We propose a compiler back-end that selects operation pattern and generates efficient code. Comprehensive experi-
mental results show that the average dynamic instruction count is reduced by over 25%, and the total energy is reduced by 15.8% compared to the RISC baseline. When high performance is required, the proposed architecture is able to achieve a speed-up of 1.14× by introducing multi-cycle SFU. The results demonstrate that the proposed solution achieves a good balance between flexibility and energy efficiency.

Then we propose a scalable wide SIMD processor architecture and a compiler for it that supports OpenCL. As the RF consumes even larger portion of energy in SIMD architecture than in scalar architectures, the processing elements (PEs) in the proposed architecture use the explicit bypassing. Features that enable the mapping of programs written in the OpenCL parallel language are added to the proposed architecture. The design of a compiler that compiles OpenCL program and optimizes memory mapping for the proposed architecture is presented. Detailed experiments are carried out on processors with different configurations. The results show that the proposed architecture and compiler are able to achieve substantial improvement in both performance and energy consumption for OpenCL programs. In a 128-PE processor instance, the proposed architecture is able to achieve over 200 times speed-up and reduce the energy consumption by over 49.5% compared to a basic RISC processor.

Last but not least, a complete hardware-software co-design framework for a configurable accelerator is proposed. The framework consists of an RTL generator, a compiler with runtime libraries, and a cycle-accurate simulator. The RTL generator generates implementation of an accelerator base on the proposed wide SIMD architecture for different target technologies, including ASIC and FPGA. The compiler can compile OpenCL program for the accelerator. The cycle-accurate simulator with debugging support is able to perform fast simulation for architectures with different configurations. The proposed framework is used for different applications, which demonstrates that it can be used to perform exploration in designing energy efficient processor for streaming applications within a heterogeneous multi-core system.
# Contents

Abstract

1 Introduction
   1.1 Trends in Embedded Systems ................................. 2
   1.2 Challenges in Energy Efficient System Design .......... 5
   1.3 Problem Statement .......................................... 7
   1.4 Thesis Outline and Contributions ...................... 8

2 Background
   2.1 Power Consumption and Energy Efficiency ............... 11
   2.2 Embedded Processor Architecture ..................... 14
   2.3 Code Generation and Programming Languages .......... 17
   2.4 Energy Awareness in Compilers ......................... 20
   2.5 Summary .................................................. 23

3 Explicit Datapath Architecture
   3.1 Reducing Register File Accesses ....................... 27
   3.2 Architectures with Explicit Datapath ................. 28
   3.3 Code Generation for Explicit Bypassing ............ 32
   3.4 Experimental Results ................................. 39
   3.5 Related Work ........................................... 46
   3.6 Summary ................................................ 48

4 Energy Efficient Flexible Special Instructions
   4.1 Special Operation Patterns ............................... 51
   4.2 Flexible Special Instructions in a RISC Processor with Compact ISA ............... 54
   4.3 Code Generation for Reconfigurable SFU ............ 59
   4.4 Experimental Results .................................. 64
   4.5 Related Work ........................................... 70
   4.6 Summary ................................................ 72
## Table of Contents

5 Exploiting Data-Level Parallelism 73

5.1 Low-Energy Wide SIMD Architecture 74
5.2 Code Generation for Wide-SIMD 78
5.3 Experimental Results 84
5.4 Related Work 87
5.5 Summary 90

6 Architecture and Code Generation Framework 93

6.1 Automatic RTL Implementation Generator 94
6.2 Accelerator Architecture Template 95
6.3 Hardware-Software Co-Design Framework 95
6.4 Case Study 98
6.5 Summary 99

7 Conclusions and Future Work 101

7.1 Conclusions 101
7.2 Future Work 103

References 105

A Baseline RISC Instruction Set Architecture 119

A.1 Instruction Set Architecture Overview 119
A.2 Registers 120
A.3 Instruction Format 120
A.4 Supported Operations 122
A.5 Summary 125

Acknowledgments 127

Curriculum Vitae 129

List of Publications 131
An *embedded system* is a computer system lodged in other devices that is designed to perform one or a number of dedicated functions. Though the presence of the computers is not immediately obvious in some cases, embedded systems are the fastest-growing portion of the computer market [34, 72]. Market research shows that the worldwide market for embedded technology was $113 billion in 2010, and is expected to reach $158.6 billion by 2015 [72]. Figure 1.1 illustrates the main application domains of modern embedded systems. It is quite obvious that embedded systems are widely used in almost every aspect of our daily lives. They are embedded in all kinds of systems, ranging from small mobile devices such as smart phones, to large machineries like airplanes [68]. With the fast-advancing technologies, embedded systems are becoming more and more powerful and they are playing a key role in improving life quality and productivity.

Unlike general computing systems such as desktop computers, an embedded system is usually designed as part of a larger system. Consequently, the design of such systems has to meet

---

**Figure 1.1**: Major application domain of embedded systems [68].
not only the functional requirements that specify what the system shall do, but also the non-functional requirements that specify what/how the system shall be. The non-functional requirements impose a series of constraints to the system design in different aspects. For example, in mobile communication devices have to process wireless signals in real-time, with very limited power and silicon area footprint. Therefore, the design methodology for embedded systems is very different from the one for general computers, even though they may share technologies and components.

The focus of this thesis is on embedded system for mobile devices. In such devices, embedded systems are usually contained in a very compact packaging, with limited power sources and cooling facilities. These constraints together shape embedded systems into the characteristics of domain specific, real time, low power and energy, low cost, and small area. Energy efficiency is one of the most important requirements among them.

The thesis tackles the problem of energy efficiency of embedded processors, by providing solutions in both architecture and code generation aspects. We focus on reducing the overhead in the datapath of embedded processors and exploiting data-level parallelism. A framework is also proposed to effectively utilize the methods described in this thesis for embedded streaming applications.

The remainder of this chapter proceeds as follows. Section 1.1 discusses the trends in embedded systems and embedded streaming applications. The major challenges in designing efficient embedded architectures are Section 1.2. Section 1.3 clarifies the research problems this thesis attempts to solve. The structure of the thesis and its main contributions are stated in Section 1.4.

1.1 Trends in Embedded Systems

There is an increasing demand of running high performance applications on mobile devices. On the other hand, we often see that more and more constraints are imposed on these devices in order to improve user experience. Table 1.1 shows the specification of the Apple iPhones of five different generations. It is clear that Moore’s Law has been boosting the development of more powerful processors, allowing users to run more complex applications. For example, compared to the first generation iPhone, the iPhone 5S has a much more complex CPU (dual superscalar cores vs. single scalar core, 1.3 GHz vs. 412 MHz) and GPU (four cores vs. single cores, 200 MHz vs. 103 MHz). And yet due to CMOS technology shrinking, the area only increases by 41.6%. Another important trend is that the battery capacity of such devices is not growing nearly as fast as the processing power [145]. On top of that, the growing demand for light-weight devices is making it more difficult to increase the battery capacity. Table 1.1 shows that unlike other components, the battery capacity in different generations of iPhone stays more or less at the same level. As a result, the battery capacity of these devices is limited, which significantly affects the user experience.

These trends require the embedded system designers to develop high-performance computing systems with limited power budgets. Therefore design for energy efficiency is the most
**Table 1.1:** Specification of iPhone of different generations[159]

<table>
<thead>
<tr>
<th></th>
<th>iPhone 3G</th>
<th>iPhone 3GS</th>
<th>iPhone 4</th>
<th>iPhone 4S</th>
<th>iPhone 5</th>
<th>iPhone 5S</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Released</strong></td>
<td>2007.7</td>
<td>2008.7</td>
<td>2009.6</td>
<td>2010.6</td>
<td>2011.10</td>
<td>2012.9</td>
</tr>
<tr>
<td><strong>Main SoC</strong></td>
<td>ARM11</td>
<td>Cortex-A8</td>
<td>Cortex-A8</td>
<td>Swift</td>
<td>Cyclone</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(ARMv6)</td>
<td>(ARMv7)</td>
<td>(ARMv7)</td>
<td>(ARMv7)</td>
<td>(ARMv8)</td>
<td></td>
</tr>
<tr>
<td><strong>Die Size</strong></td>
<td>@90nm</td>
<td>@65nm</td>
<td>@45nm</td>
<td>@45nm</td>
<td>@32nm</td>
<td>@28nm</td>
</tr>
<tr>
<td><strong>CPU</strong></td>
<td>72mm²</td>
<td>72mm²</td>
<td>53mm²</td>
<td>122mm²</td>
<td>97mm²</td>
<td>102mm²</td>
</tr>
<tr>
<td></td>
<td>@90nm</td>
<td>@65nm</td>
<td>@45nm</td>
<td>@45nm</td>
<td>@32nm</td>
<td>@28nm</td>
</tr>
<tr>
<td><strong>GPU</strong></td>
<td>MBX Lite</td>
<td>SGX535</td>
<td>SGX535</td>
<td>SGX543</td>
<td>G6430</td>
<td></td>
</tr>
<tr>
<td></td>
<td>103MHz</td>
<td>150MHz</td>
<td>800MHz</td>
<td>200MHz</td>
<td>266MHz x3</td>
<td>200MHz x4</td>
</tr>
<tr>
<td><strong>Display</strong></td>
<td>480×320 (163ppi)</td>
<td>960×640 (326ppi)</td>
<td>1136×640 (326ppi)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Battery</strong></td>
<td>1400mAh</td>
<td>1150mAh</td>
<td>1219mAh</td>
<td>1420mAh</td>
<td>1432mAh</td>
<td>1440mAh</td>
</tr>
<tr>
<td><strong>Weight</strong></td>
<td>135g</td>
<td>133g</td>
<td>135g</td>
<td>137g</td>
<td>140g</td>
<td>112g</td>
</tr>
</tbody>
</table>

**Figure 1.2:** H.264/MPEG-4 AVC codec pipeline.

important aspect of designing embedded computing systems. It is crucial to optimize system design based on application characteristics. In Section 1.1.1, we introduce a type of application that is becoming increasingly popular in embedded systems, namely, streaming applications. To meet the requirement of different applications within limited budget, embedded system designers often exploit heterogeneity in System-on-Chips (SoCs). Section 1.1.2 discusses the trend of heterogeneous in more details.

### 1.1.1 Embedded Streaming Applications

Many applications in embedded systems perform the same or similar operations on regular sequences of data, which can be categorized as streaming applications [148, 149]. For example, on a smart-phone, one can easily find many streaming applications that are essential to the functionality of the system, like wireless communication, high-definition video/audio codecs and 3D graphics rendering.

A streaming application usually contains a set of independent processing stages. Figure 1.2 depicts the structure of a H.264 video codec, which is a typical streaming application. The different stages or the codec form a pipeline that is used to encode or decode videos frames. A frame in a H.264 stream can be an I-frame, a P-frame or a B-frame. When processing a video stream, the behavior of the codec remains largely unchanged for each type of frames [73]. Many
embedded streaming applications with high computation demand have structures organized in a similar fashion [148]. The streaming processing structure in these applications creates lots of optimization opportunities, such as pipelined parallel execution of different parts, and exploiting data level parallelism in certain stages [47]. More importantly, streaming applications are among the most performance demanding applications in embedded systems. The rapid introduction of these applications becomes an important driving force for the development of new embedded technologies. So this thesis focuses on developing energy efficient architectures and compilation techniques for streaming applications.

1.1.2 Heterogeneous Systems

As the performance requirements of applications keep increasing, multi-processor system-on-chips (MPSoCs) have inevitably become the standard choice for embedded devices [15, 82]. In MPSoCs, applications are able to exploit parallelism to improve performance and reduce energy consumption. The diversity in emerging applications makes asymmetric or heterogeneous multi-core design an ideal choice for building efficient systems [61]. Figure 1.3 depicts an example of heterogeneous MPSoC, which consists of a general purpose processor (GPP), a graphics processing unit (GPU), a hardware accelerator, an application specific instruction set processor (ASIP) and other peripheral IPs. By introducing heterogeneity, each application can be mapped to the part of the system that is most suitable for it, further increasing the energy efficiency.

Heterogeneous SoCs have been widely used for both general purpose computers like the AMD APU [18], and embedded systems like TI OMAP [30] and NVIDIA Tegra [100]. Different types of application-specific cores are used in these SoCs to improve the performance and efficiency. Recently, heterogeneous implementation of the same instruction set architectures (ISAs) becomes a popular choice for power optimization in embedded systems. Example of such systems include NVIDIA’s Variable-SMP and ARM’s big.LITTLE architectures [84, 101]. On a same-ISA heterogeneous multi-core system, an application can run on a fast but power hungry core, or on a slow but low power core, depending on the demand and resource availability.
1.2 Challenges in Energy Efficient System Design

As discussed in Section 1.1, low power is usually a distinguishable feature and one of the main challenges in embedded systems. Figure 1.4 shows the energy efficiency requirements for two important applications on smart-phones, wireless communication and high-definition video applications. To handle emerging wireless standards like 4G LTE and H.264/H.265 video codecs on a single computing device, energy efficiency beyond 1 pico-joule per operation is required [15, 161].

To achieve high efficiency, implementing applications in dedicated Application Specific Integrated Circuits (ASICs) is a tempting choice. However, the lack of flexibility of such design cannot meet the requirement of the fast-changing demands of modern embedded devices. For example, a baseband processor for mobile phones needs to support multiple wireless standards, with the possibility of adding newly developed ones. To achieve cost-effectiveness and have a short time-to-market, a programmable platform is required [15]. All these requirements call for an energy efficient, programmable embedded processing platform that is able to bridge the gap between ASICs and processors.

More details are discussed in the remainder of this section. We discuss the challenges in improving the efficiency of processors and utilizing them in heterogeneous MPSoC in Section 1.2.1 and Section 1.2.2, respectively. As most processors are implemented in CMOS technology, we discuss designing low power and low energy circuit in Section 1.2.3.

1.2.1 Processor Computational Efficiency

The energy efficiency gap between general purpose processors (GPPs) and application specific integrated circuits (ASICs) is huge. A GPP may consume a few orders of magnitude more energy than an ASIC that performs the same computation [55]. One of the biggest contributors to this gap is the overhead associated with executing instructions from applications running on the processor. The overhead usually consumes much more energy than the actual useful computation [11]. Hence reducing the computation overhead is one of the biggest challenges...
in designing an energy efficient processor architecture.

In processors, a substantial part of the overhead comes from storing temporary results and transporting them between storage components and function units (FUs). Within the processor datapath, storage components like the register file (RF) are among the biggest energy consumers [10, 157]. Hence reducing the energy consumption of these components is one of the key steps towards an efficient processor architecture.

Another major source of inefficiency in processors is the control overhead. In a processor, the generic resources are controlled by instructions to perform computation for different applications. Compared to ASICs, such flexibility leads to design with much higher overhead. While this is an inherent drawback of programmable processing platforms, it is possible to find a balance between flexibility and efficiency. To find the balance, it requires optimization not only in architecture and hardware, but also in software, particularly, the compiler.

**Energy-Aware Compilation**

For a programmable platform, the compiler is always one of the most crucial system softwares, as it greatly improves the productivity of software development. Whether a processor architecture can be fully exploited heavily depends on the quality of the code generated by its compiler. Compiler optimization may have large impact on the power and energy consumption at different levels of the processor architecture [89]. Traditionally, compilers use performance metrics for optimization. While there is a high correlation between performance and energy efficiency, compilers need to be more aware of power and energy related metrics and have better models for analysis and optimization, especially for embedded systems [162].

In embedded systems where resources are limited, it is common to shift the burden of optimization to development time as much as possible. Such architecture changes often require software to be aware of the architectural details to achieve high efficiency. For example, in embedded digital signal processing, the Very Long Instruction Word (VLIW) architecture, which relies on the compiler to discover Instruction-Level Parallelism (ILP), is often the preferred choice, rather than super-scalar architecture that detects and exploits ILP dynamically by the hardware [59]. Therefore a compiler that is capable of generating efficient code is a key in low-power embedded processor architecture design.

### 1.2.2 Adapting Processors to Heterogeneous MPSoC

As discussed in Section 1.1.2, heterogeneous MPSoC is very important for efficient embedded systems. Therefore, it is crucial to adapt processors to heterogeneity of the system when designing embedded processor architectures and tools. In addition to performing computation efficiently, a low-power processor should be able to be integrated into a heterogeneous system easily. Having an interface for efficient integration into heterogeneous MPSoC is very important. On top of that, a compiler and runtime environment that support languages that are designed for heterogeneous parallel computing systems, such as OpenCL, are required [67, 116].
1.2.3 Low Power Digital Circuit Design

Embedded processors are usually built on CMOS digital circuit technologies. Therefore it is important to combine the architecture-level with circuit-level techniques. Some techniques like the adaptation of new transistor technologies are transparent to architecture and system level [127]. But to fully exploit the low power potential, architectures and system softwares including operating systems and compilers that can utilize these techniques are required, especially for techniques like dynamic voltage and frequency scaling (DVFS) [63, 109].

1.3 Problem Statement

The goal of this thesis is to develop architecture and code generation techniques that can achieve high energy efficiency in embedded systems. However, it is not realistic to have one single solution for the vastly varying application domains of embedded systems. So we focus on streaming application, which, as discussed in Section 1.1.1, is an important type of applications, especially in emerging mobile devices. This thesis tackles some of the challenges discussed in Section 1.2. By focusing on streaming applications, which have common characteristics such as relatively regular control structure and high level of data level parallelism, more effective methods to improve the energy efficiency can be developed.

Efficient Datapath and Code Generation for Embedded Systems

One of the main reasons that a processor is inefficient compared to application specific circuitry is that a lot of energy is spent on moving data between function units (FUs) that perform actual computation and storage elements that store the temporary results of the computation. In particular, the register files (RFs) that exist in most processor architectures often consume a substantial portion of the core energy [43, 56, 161]. Observe that in many typical streaming applications, most variables are local and are used only for very few times [56]. So it is possible to reduce accesses to the RF by using an explicit datapath that allows the software to specify direct communication between FUs. For example, the Transport-Triggered Architecture (TTA) can substantially reduce the traffic between FUs and RFs by allowing software to have fine-grained control over the datapath [29]. However, it also suffers from low code density, which is potentially quite harmful for energy efficiency, especially for processors that have to run applications from different domains as domain-specific optimization cannot be used [36]. Finding the right balance between flexibility and control overhead is crucial for improving the energy efficiency of an explicit datapath architecture. And as softwares have fine-grained datapath control, the compiler plays an important role in explicit datapath architectures.

Another important way to improve datapath efficiency is to support special instructions that execute complex patterns consist of more than one basic operations. In Application Specific Instruction Set Processors (ASIPs), it is common to use special instructions for improving performance and efficiency [76, 90]. Commercially successful examples include Tensilica Xtensa [120] and IMEC ADRES [96]. However, it is quite a challenge to apply similar ideas of special function unit (SFU) to generic embedded processor architectures, because supporting
arbitrary complex operation patterns may incur huge overhead that diminishes the energy gain. Previous studies primarily focus on improving performance [25, 26]. To achieve high energy efficiency, the support for special instructions needs to have low energy overhead, while still being able to support applications from different domains. The main challenges are i) choosing the appropriate set of operation patterns and designing an SFU that executes them; ii) integrating the SFU into the processor datapath efficiently; iii) automatically discovering the supported patterns and using the SFU in the applications.

Exploit Data-Level Parallelism to Improve Energy Efficiency

Streaming applications usually possess an abundant amount of Data-Level Parallelism (DLP). DLP can be exploited by Single Instruction Multiple Data (SIMD) architectures, in which multiple processing elements (PEs) execute the same instruction on different data items simultaneously [59]. SIMD is inherently energy efficient as a substantial part of the control overhead is amortized over PEs. It has been widely used in main-stream general-purpose processors and GPUs [32, 79, 112, 124]. An embedded processor based on a wide SIMD (> 64 PEs) architecture can substantially improve the energy efficiency by amortizing the overhead over a larger number of PEs. Examples of such processors include Xetal [1] and IMAP [85]. To efficiently utilize wide SIMD architectures, the software needs to exploit a high degree of DLP. This is challenging as most application are specified in sequential languages like C. Though there are auto parallelization methods that extract parallelism in sequential programs, they cannot fully exploit the potential of such highly parallel architectures. For scalability reasons, wide SIMD processors tend to be much less flexible compared to general purpose processors, which makes it more difficult to program.

Framework for Design with Efficient Embedded Processor

As discussed in Section 1.1, modern mobile devices rely on MPSoCs that contain various types of components. Designing an efficient embedded processor is not only about improving the efficiency of the processor itself, but also about how to integrate it into a heterogeneous system. To achieve that, it requires interfaces and tools that enable software-hardware co-design for emerging streaming applications.

1.4 Thesis Outline and Contributions

To addresses the problems stated in Section 1.3, this thesis proposes methods in architecture and code generation for designing energy efficient embedded processor targeting streaming applications. Chapter 2 provides essential background information in power consumption of digital circuits, code generation and energy awareness in compilers. The thesis proceeds with four chapters that presents the main contributions:

Explicit Datapath and Code Generation

The register file (RF) is one of the most frequently used, and most power-hungry components in a processor. And it consumes a considerable amount of energy. By introducing an explicit
datapath that enables fine-grained control in the software, the energy consumption of the RF can be reduced dramatically. Efficient code generation for such architecture is key to achieve high energy efficiency for the whole processor. In this work we propose a new compiler back-end for explicit bypassing. The compiler includes new algorithms to schedule instructions such that most of the unnecessary RF accesses are eliminated, while the performance is unaffected.

Experiments show that on the proposed architecture, 70% of the RF accesses are eliminated. Energy analysis shows that the proposed method is able to reduce the core energy consumption by an average of 15%. The result indicates that it achieves a good trade-off between flexibility and efficiency compared to other architectures with explicit datapath, like TTA [36] (Chapter 3).

Flexible Special Instructions in Compact Processor Architecture
A method to support flexible operation-pair patterns in a processor with compact instruction set architecture (ISA) is proposed. An explicit datapath combined with a partially re-configurable instruction decoder is used to reduce the overhead of supporting the function unit that can execute arbitrary operation pairs. A compiler design is proposed and implemented. The compiler back-end selects operation patterns and generates efficient code for the proposed architecture.

Comprehensive experiments are carried out. The results show that the average dynamic instruction count is reduced by over 25%, and the total energy is reduced by an average of 15.8% compared to the RISC baseline. When high performance is required, the proposed architecture is able to achieve an average speed-up of 13.8% with 13.1% energy reduction compared to the baseline by introducing a multi-cycle SFU (Chapter 4).

Efficient Code Generation for Wide-SIMD Processor
We propose a scalable wide SIMD processor architecture with PEs that uses the explicit datapath described in Chapter 3. The proposed architecture has features that enable the mapping of programs written in the OpenCL parallel language. The design of a compiler for the wide-SIMD architecture is proposed. The compiler compiles OpenCL program and optimizes memory mapping for the proposed architecture. Detailed experiments are carried out. The results show that the proposed architecture and compiler are able to achieve substantial improvement in both performance and energy consumption for OpenCL programs. In a 128-PE processor, the proposed architecture is able to achieve over 200 times speed-up and reduce the energy consumption by over 49.5% compared to a basic RISC processor (Chapter 5).

Co-Design Framework for Efficient Processor Architecture
A complete design framework is proposed. The framework is capable of generation implementation of the proposed wide-SIMD processor for different targets, including ASIC and FPGA. The software toolchain, including compiler, runtime libraries and simulator provides an efficient design environment for developing streaming applications on the proposed architecture within a heterogeneous multi-core system (Chapter 6).

Last but not least, Chapter 7 concludes the findings of this thesis and gives recommendations to future research directions in this topic.
Background

As discussed in Chapter 1, designing an energy efficient computing system requires efforts at different levels. The focus of this thesis is on the joint optimization of architecture and code generation. This chapter introduces essential background knowledge related to the work of this thesis.

The remainder of this chapter is organized as follows. Section 2.1 discusses general concepts of power consumption and energy efficiency of digital circuits. Section 2.2 introduces different types of processor architectures in embedded systems, as well as issues related to energy efficiency. Section 2.3 gives an overview of modern compiler design and code generation techniques for embedded systems. Section 2.4 discusses the role compilers play in low-energy embedded system design. Lastly, Section 2.5 concludes this chapter.

2.1 Power Consumption and Energy Efficiency

In CMOS circuits, most power is dissipated when the gate output changes. The power consumption of a clocked CMOS digital circuit is determined by Equation 2.1. The first term is the dynamic power dissipation, where $\alpha C$ is the switched capacitance of the circuit, $V_{dd}$ is the supply voltage, $f$ is the operating frequency. The second term is the leakage power, where $I_{\text{leak}}$ is the leakage current.

$$P = \alpha CV_{dd}^2 f + I_{\text{leak}}V_{dd} \tag{2.1}$$

There are many circuit-level techniques to reduce the power consumption of CMOS circuits [21, 123]. Dynamic power is typically the dominant source of power consumption. The $V_{dd}^2$ factor suggests that $V_{dd}$ has the greatest impact on dynamic power. In practice, voltage and frequency scaling (VFS) is an effective method of reducing the dynamic power [21]. In recent technology node, however, the leakage power is becoming more and more significant, especially for structures like memories in which most parts of the circuit stay idle for most of the time [81]. Such trend requires better trade-offs between dynamic and static power when doing frequency and voltage scaling [57, 121, 122].

Aside from techniques that directly reduce the power consumption of the circuit, it is also
important to increase the portion of energy that is spent on useful computation, i.e., improve the computational efficiency. The remainder of this section introduces the concept of intrinsic computational efficiency and issues that affect the energy efficiency of digital circuits.

2.1.1 Computational Efficiency of Digital Circuits

The intrinsic computational efficiency (ICE) represents the potential raw computational efficiency of CMOS circuit [131]. Figure 2.1 depicts the ICE of different CMOS technologies and the peak performance of various processors\(^1\). The stair shape of the ICE curve mainly comes from the voltage reduction in new technologies. The lower line is the projected computational efficiency of general purpose processors (GPP). Although the efficiency of modern GPPs is able to scale beyond this line, they are still far from the ICE. In a typical 32-bit RISC-like processor, an *add* instruction consumes *10 times* energy than the energy spent on a 32-bit adder [11]. On top of that, processors have other overhead in order to perform useful computation, such as instruction and data memory accesses. All these results in a huge gap between the ICE of the circuit and the efficiency of programmable processors, as illustrated in Figure 2.1. Even with the most efficient processors, there is still a huge gap between the requirement of emerging mobile applications, such as 4G wireless communication and HD video applications, and the energy efficiency of GPPs.

\(^1\)In [131], the silicon ICE is calculated based on a circuit with a 32-bit full-adder and a register in a reference technology.
Figure 2.2: Trade-off between efficiency and flexibility.

Figure 2.3: Generic view of a processor.

2.1.2 Overhead in Programmable Processors

Figure 2.2 illustrates the trade-off between flexibility and efficiency in computing system design. In general, dedicated hardware (ASIC) has the efficiency that is the closest to ICE among all solutions, but only for a very limited set of applications. On the other hand, general purpose processor (GPP)\(^1\) provides the best flexibility as it can run virtually any application, at the cost of much lower performance and much higher energy consumption compared to ASICs. There are other types of computing systems offering different trade-offs between efficiency and flexibility. Application-Specific Instruction-set Processor (ASIP) adapts the Instruction Set Architecture (ISA) for a specific set of applications, which can bring the processor efficiency close to ASIC for the target applications [55]. Field Programmable Gate Array (FPGA) offers high flexibility with reconfigurable structured hardware blocks. Graphics Processing Unit (GPU) is designed for graphics rendering. In recent years, the introduction of General-Purpose GPU (GPGPU) also enables the use of GPU in other application domains, especially in scientific computing [91, 98].

For a programmable architecture, the gap between ICE and its efficiency mainly comes from the overhead it requires to provide the required flexibility. Figure 2.3 shows an abstract structure of a processor. To some extent, all components except the datapath can be considered as the overhead of the computation. Application-specific optimizations, including architecture and compiler optimizations, are known to be effective for improving the efficiency dramatically [55, 130]. In this thesis, we focus on developing techniques that can reduce the overhead in processors for applications in different domains, which offer flexibility close to GPP but more efficient.

\(^1\)In this thesis Digital Signal Processor (DSP), such as the TI C64 series [65], is also considered as GPP. Because although not as flexible as GPPs, DSPs are capable of handling computation in various applications.
2.2 Embedded Processor Architecture

A processor is a hardware unit that carries out the instructions of a software program. It is usually one of the central components within a computer system. In modern embedded systems, the processor is also one of the key building blocks. There are various types of processors that are designed to meet different requirements. Processors based on the Reduced Instruction Set Computing (RISC) concept are widely used in many embedded systems. We introduce the basic concepts of RISC in Section 2.2.1. Exploiting parallelism in application is important for not only performance, but also power and energy consumption of a processor. Section 2.2.2 gives an overview of how parallel computing can impact low power processor design, and different ways to exploit parallelism.

2.2.1 RISC Architectures

Reduced Instruction Set Computing (RISC) is a common processor design strategy in embedded systems [108]. The following features are found in a typical RISC architecture:

- Instruction length is fixed for all instructions, with only a few different formats, making the instruction fetching and decoding simple.
- Only a limited number of simple data types are supported (for example, integer and floating point). Complex types like string are not directly supported.
- Only load/store instructions may access memory, which simplifies the pipeline design as the logic for dealing with the memory access delay is isolated.
- Only a few simple addressing modes are supported. Complex addressing is performed via sequences of arithmetic, load/store operations.

Figure 2.4 depicts the datapath of a classical RISC processor. To improve the performance and exploit parallelism among different instructions, the datapath of the processor is divided into 4 stages, namely, Fetch, Decode, Execute and Write-back (more or fewer stages can be implemented depending on the throughput requirements).

The concept of RISC is simple and yet effective: it makes the hardware much simpler compared to Complex Instruction Set Computing (CISC) architectures and also eases the de-
Pipeline Hazard and Bypassing

A pipelined processor exploits Instruction-Level Parallelism (ILP) by executing different instructions in each stage. The result of each instruction is stored in the register file (RF) after the write-back stage. However, when executing an instruction with data dependency, pipelining creates data hazards that have to be handled. Figure 2.5 shows an example of a data hazard. In this example, the results of the first two multiply instructions (r3 and r4) are needed before they are actually stored to the RF. To avoid pipeline stalls (i.e., inserting bubbles into the pipeline), processors usually introduce a bypassing network (also called forwarding network) that automatically detects and resolves data hazards by transferring results of previous instructions that are in the pipeline. The effect of such a network is shown in Figure 2.4 and Figure 2.5. In Chapter 3, we will show that with proper architectural modifications and compiler support, the bypassing network is also capable of improving the energy efficiency.

2.2.2 Parallelism and Low Power

The amount of parallelism in a program is the amount of operations that can be executed simultaneously. Traditionally, exploiting parallelism is effective for improving performance. In addition, parallel computing can also reduce power consumption. As illustrated in Figure 2.6, when more parallelism is exploited, the same amount of computation can be done at lower frequency, enabling a lower supply voltage. As indicated in Equation 2.1, voltage has great impact on the dynamic power consumption of CMOS circuits. Therefore exploiting parallelism is important for designing energy efficient computing systems.

---

1The assembly used here is similar to the OpenRISC assembly [102].
Architecture for Exploiting Instruction-Level Parallelism

Instruction-Level parallelism (ILP) is a measure of the number of operations that can be executed simultaneously in a program. There are two typical types of processor architectures for exploiting ILP:

- **Superscalar** architecture: the processor dynamically detects available ILP in the sequential instruction streams in the programs and issues multiple operations in each cycle.
- **Very Long Instruction Word (VLIW) architecture**: the compiler is responsible for discovering ILP and scheduling multiple operations in each cycle. The processor does not need to perform any runtime checking.

In embedded systems, many processors exploit ILP using VLIW architectures [40]. Superscalar processors are also used in some modern embedded devices where high-performance general-purpose processing is required, for example, the ARM Cortex-A series processors [112], but in general, the cost of dynamic scheduling in superscalar is very high, making it a less efficient choice [83]. On the other hand, a SuperScalar ISA allows the implementation to adapt to the new IC technologies easier, without modifying the ISA. While for a VLIW ISA, in which usually pipeline stages and delays are fixed, the implementation in new IC processing technology may be sub-optimal. Intel Itanium tried to define an architecture based on the VLIW concept that can scalable over multiple technology nodes [138, 94]. However, Itanium turned to be not successful in the mainstream market for many reasons.

Architecture for Exploiting Data-Level Parallelism

In many applications, it is common that a significant portion of the program is performing similar (in many cases the same) but independent computation on different data items. Such applications contain substantial amounts of Data-Level Parallelism (DLP) [59]. It is possible to exploit DLP by issuing multiple identical instructions for different data items in VLIW or superscalar processors. However, a more efficient strategy is to exploit DLP by adapting single-instruction-multiple-data (SIMD) processing, because in SIMD, the cost of instruction fetching, decoding and large portion of the control logic are shared by a number of PEs. SIMD extensions have been used in general purpose processors, such as SSE [124], Altivec [32] and AVX [39]. It is one of the reasons that modern general purpose processors are able to scale beyond the lower line in Figure 2.1.
As shown in Figure 2.1, the processors that achieve high efficiency (e.g., Xetal-II [1] and AnySP [161]) all contain wide SIMD processing capability. Recent GPU architectures also achieve high energy efficiency by using SIMD processing [104, 79]. The Imagine architecture is an example that combines clustered-VLIW and SIMD [80]. Figure 2.7 depicts the architecture of an Imagine stream processor. The Imagine processor is able to achieve very high efficiency compared to other architectures with the same technology.

2.3 Code Generation and Programming Languages

A compiler is a program that translates source code in a programming language into another computer language [3]. In this thesis, the compiler is referred to as the tool that translates source code in high-level languages into machine code of target architectures. A compiler plays a key role in modern software development as it enables machine-independent programming, as well as the use of high-level languages like C/C++. The productivity of software development is dramatically improved by compilers. The basic structure of modern compilers is introduced in Section 2.3.1. Section 2.3.2 presents an overview of compilation and programming languages for parallel architecture.

2.3.1 Modern Compiler Architecture

The process of a modern compiler can be divided into three phases:

- A front-end checks the syntax and semantics of the source programs in high-level languages. It produces the intermediate representation (IR) of the source programs. The IR used in a compiler is usually independent of the source language and target architecture.
- A middle-end analyzes and transforms the IR to optimize the program for various objectives. Due to the independent nature of the IR, most of the middle-end usually does not rely on information of specific programming language or target architecture. For example, inline expansion, dead code elimination and loop transformations are usually done in the middle-end.
• A back-end generates machine code for the target architecture from the IR. The back-end performs machine-dependent transformations and translates IR into machine code. Typical back-end phases include instruction selection, scheduling, and register allocation.

Figure 2.8 depicts the generic structure of a compiler with the afore-mentioned three phases. With such structure, it is relatively easy to construct compilers for a new programming language or a new architecture: only the corresponding parts in the front-end or back-end need to be modified. The middle-end, which contains most common compiler analysis and optimization passes, can be reused.

The LLVM framework has the same structure as shown in Figure 2.8 [87, 92]. The IR used by the LLVM middle-end, called LLVM-IR, is a Static Single Assignment (SSA) based lightweight, low-level and typed, universal IR [147]. The framework has a front-end called Clang that translates C-family languages (e.g., C, C++ and OpenCL C) into LLVM-IR [23]. The code generation framework in LLVM is flexible enough to support different types of targets including CPUs and GPUs [92]. In this work, the LLVM open-source compilation framework is used for the compiler development, as its modular design makes it more suitable for supporting new targets compared to alternatives like GCC.

Traditionally, compilers compile programs before they are executed, which is called static compilation. It is also possible to compile programs at runtime, which is called dynamic compilation or Just-In-Time (JIT) compilation [9]. Although static compilation and dynamic compilation have different requirements, many code generation techniques can be used for both. For example, the LLVM framework is used to build both static and dynamic compilers.

2.3.2 Programming and Compiling for Parallel Architectures

An important aspect in designing a new architecture is how to efficiently develop programs for it. Traditionally, most programs are specified in sequential programming languages like C. For most parallel architectures, the amount of parallelism in sequential programs is insufficient for effectively utilizing the resources. Automatic parallelization methods have been proposed, some of which are used in production compilers. For example, recent research works proposed extracting and exploiting parallelism in sequential programs using Polyhedral model [49, 97, 99]. However, for highly parallel architectures such as GPUs, the state-of-the-art auto parallelization still cannot fully exploit the potential of such architectures [155]. Parallel program-
ming languages are able to improve the programmability and efficiency for massive-parallel architectures. NVIDIA introduced CUDA that enables programming GPUs for general purpose computing applications [91, 98]. In this work, we use OpenCL, a standardized parallel language for massive-parallel SIMD architectures, because the features of OpenCL make it suitable for SIMD execution. The reminder of this subsection gives a comprehensive introduction about OpenCL.

OpenCL Parallel Programming Language

OpenCL (Open Computing Language) is an open standard for developing parallel programs on various heterogeneous platforms [116]. GPGPUs are among the most important target architectures of OpenCL [111, 160], but many different types of architectures can also efficiently support OpenCL, such as general purpose CPUs [74], FPGAs [103] and ASIPs [67]. The conceptual platform architecture assumed by OpenCL is shown in Figure 2.9. A platform contains a number of compute devices, in which the processing elements (PEs) are grouped into compute units. The memory is divided into four different address spaces: private, local, global and constant. The OpenCL standard defines:

- a C-based language called OpenCL C that used to define kernels for performing computation on compute devices;
- a set of APIs in standard C for invoking the kernels from the host and transferring data between the host and compute devices.

In an OpenCL kernel, the workload is divided into work-groups. Each work-group consists of a number of work-items with different indices. Figure 2.10 illustrates the index space of an OpenCL kernel. The index space supported in OpenCL is called an NDRange, which is an

![Figure 2.9: Conceptual OpenCL platform architecture and address spaces.](image-url)
Figure 2.10: The index space of a 2-D OpenCL kernel. The invocations of kernels are controlled by command queues, which enables task-level parallelism.

N-dimensional index space, where N is one, two or three. Work-groups are assigned a unique work-group ID with the same dimensionality as the index space used for the work-items. In OpenCL kernel semantics, the work-items in a given work-group execute concurrently on the processing elements of a single compute unit. Different work-items can only synchronize by calling synchronization functions explicitly. So different work-items of the same kernel can be executed in parallel between explicit synchronization points. Figure 2.11 depicts the Single Program Multiple Data (SPMD) execution model in OpenCL. In this model, the kernels are mapped onto the compute devices. The host controls the overall program execution using a structure called command queue, which contains memory copying, kernel invocation and synchronization commands. The command can be scheduled in-order or out-of-order, and execute asynchronously between the host and the devices.

This model is ideal for SIMD architectures because: i) work-items of the same kernels execute the same instruction sequence, which is easy to fit in SIMD semantics; ii) the implicit independence of work-items gives the compilers more freedom to map and schedule them on SIMD processors. Each work-item is mapped onto a processing element (PE) and each work-group is mapped onto a compute unit. The different address spaces make the analysis and mapping of communications between work-items easier for SIMD architectures.

The two most important aspects of mapping OpenCL kernels onto a processor are:

- Map and schedule work-items on the PEs of the target architecture.
- Map the different address spaces onto the memory hierarchy of the target architecture.

2.4 Energy Awareness in Compilers

Historically, compilers focus on reducing the size and increasing the speed of the generated code during transformations. Energy awareness is relatively new in compilers. In low-power system design, software techniques including energy-aware optimizations in compilers play an
important role [95]. Optimizations for energy consumption are often related to optimizations for speed [89]. However, it is not always the case, especially for embedded systems, in which the constraints and design objectives are usually quite different from general purpose computing systems like desktop computers. Therefore compilers need to perform different types of analysis and optimization [162]. Compiler design techniques can contribute to energy reduction in different ways. In the remainder of this section, we discuss some of the options to increase the energy awareness in compilers for embedded systems.

**Reducing Switching Activities**

As described in Section 2.1, circuit switching is the main cause of power dissipation. Although a compiler usually works on a relatively high level, it is able to influence the amount of switching activity to a large extent in different ways. By analysis with proper modeling, a compiler can reduce the switching activities for a program significantly. For example, by carefully arranging the instructions of an application, the switching activities on the instruction bus can be reduced. Lee et al. proposed scheduling algorithms to minimize the switching activities by reducing the Hamming distance between instructions in VLIW processors, resulting in over 20% reduction of the instruction bus activities [88]. In [137], Shao et al. proposed a loop scheduling algorithm for VLIW processor that minimizes both schedule length and switching activities, and the results showed 11.5% improvement in schedule length and 19.4% improvement in bus switching activities compared to [88]. Dimond et al. presented a technique that combines instruction coding and instruction re-ordering, which resulted in up to 74% dynamic power reduction with no performance loss [33]. Similar techniques also apply to other parts of a processor. Mehta et al. showed that by properly assigning registers to operands and arranging their order, the compiler was able to reduce the switching activities in the register file (RF) without affecting the performance, thereby reducing the register energy by 13.26% and total processor energy by 4.25% [95]. Lee et al. showed that by properly swapping the input operands of multiplication in a DSP using Booth multiplier, 3% to 6% overall energy reduction can be achieved by [89].
Energy-Aware Resource Allocation

The compiler is responsible for managing many processor resources, especially for embedded systems in which the runtime management capabilities are limited. The allocation of resources has great impact on the energy efficiency, especially for resources that consume a lot of energy; particularly the allocation of the resources in the storage system including the register file and memories is of great importance to the energy consumption. In embedded systems, scratchpad memory that is exposed to software is a popular alternative to a cache [13]. A compiler is able to allocate data to scratchpad memories to improve both performance and energy efficiency. Avissar et al. presented a scheme for managing scratchpad memory, which achieved over 40% improvement in execution time [7]. Wehmeyer et al. showed that scratchpad memory management in the compiler can result in over 20% energy saving in the memory sub-system [158]. In [8], Ayala et al. proposed a power-aware compilation technique that can allocate registers in such a way that only part of the register file was used and configure the unused part of the register file to power save mode, which leads to 65% energy reduction in the RF without performance penalty. Compiler can help to improve not only energy consumption, but also thermal distribution in the processor. Sabry et al. presented compiler techniques based on an efficient register allocation mechanism, which reduced the hot-spots in the register file by 91% the peak temperature by 11% [132].

Loop buffer, which is common in embedded processors, is as a small memory that can hold a small number of instructions (usually from a loop body). For example the TI C64x+ has software pipelined loop (SPLOOP) buffer for storing instructions of a pipelined loop [65]. By storing the most frequently executed loops in a small loop buffer instead of a conventionally large instruction memory or cache, energy can be reduced. Uh et al. show that compiler transformations can improve the number of loops that fit in the loop buffer by over 10% [151]. Jayapala et al. presented a clustered loop buffer organization and compiler optimization that resulted in 63% compared to centralized loop buffer [70].

Supporting Power Management at Compile-Time

Modern microprocessors provide various power management facilities that can be controlled by software. A power-aware compiler is able to perform power management with low runtime overhead. Dynamic Voltage and Frequency Scaling (DVFS) is an important method to manage processor computing capacity for low power systems. It is common to perform DVFS at the operating system level [109]. However, a compiler is still able to perform effective voltage scaling in many cases [164]. Wu et al. proposed a dynamic compilation framework that was able to reduce processor energy consumption by DVFS, resulting in 22% improvement in Energy-Delay-Product (EDP) [163]. Ozturk et al. proposed compiler support for a multi-processor system with multiple voltage islands [105]. Power gating is a technique to shutdown part of the circuit at runtime in order to reduce the static power [64]. It is gaining more attention as leakage power is becoming more and more significant as semiconductor technology advances. You et al. presented a framework to schedule power gating for execution units at compile time for reducing leakage power in microprocessors, which is able to reduce total energy consumption
of both in-order and out-of-order processors [165].

**Optimizing for Instruction Set Architecture Modification**

As the instruction set architecture (ISA) has great impact on the efficiency of a processor, a common approach for reducing energy consumption is to modify the ISA so that more energy efficient components in the processor can be used. The compiler has to adapt to and optimize for such modifications.

In many embedded systems, it is common to use Application Specific Instruction-Set Processors (ASIP) that optimize the ISA for a specific set of applications [130]. In these optimization, the compiler plays a central role. For example, a typical approach is to use complex instructions to reduce the cost of performing certain computation patterns, thereby improving performance and saving energy [76, 90]. In [55], Hameed et al. showed the process of optimizing an ASIP for H.264 application that leaded to a processor matches the performance of an ASIC solutions within 3x of its energy and within comparable area.

ISA modifications in general purpose processors can also be an effective method for saving energy. For example, the ELM architecture introduces small buffers for both data and instructions, achieving much higher energy efficiency than RISC processors [10, 11, 12]. As a significant portion of the communication in the ELM datapath is exposed to software, the compiler has to perform a lot of new tasks, especially for instruction scheduling [106].

### 2.5 Summary

Improving energy efficiency of a processor is a complex process that requires efforts in different aspects. The first step towards an energy efficient processor is designing a proper hardware architecture, including the choice of instruction set architecture and the ability to exploit parallelism in the target applications. And for a processor, the efficiency of its compiler is equally important, as it is directly responsible of how the processor is used.

As we have shown in Section 2.4, processor architecture and compiler design are highly correlated, especially for low-energy embedded systems. In the remaining chapters of the thesis we explore techniques in both architecture and compiler design to build energy efficient processor for streaming applications.
Explicit Datapath Architecture

Using a hierarchical storage system is a common choice in computer systems, as there is a trade-off between capacity and efficiency in most types of memories: the larger the memory, the more costly it is in terms of area, time and energy. Such hierarchical memory systems often result in a storage system pyramid as depicted in Figure 3.1. The higher in the pyramid, the closer it is to the computational resources. Therefore data can be accessed more efficiently when it is stored in a place at a high level of the pyramid. On the top of the pyramid are small buffers in the datapath, for example, pipeline registers. In most conventional processor architectures, these buffers are managed by the hardware and are not visible in the instruction set architectures (ISAs). And in a processor with such ISA, the register file (RF) is involved in almost every operation, both as the source and the destination of operand values. As a result, the RF often consumes a substantial portion of the total core energy [44, 56, 161]. Figure 3.2 depicts the power breakdown of a 4-stage RISC processor running an FIR filter, in which the RF (with two read ports and one write port) consumes 18% of the total core power. In more complex processor architecture like VLIW and SIMD, this number is even higher, because the complexity and capacity of the RF increase, while the relative size of the control path becomes smaller [157, 161].

In this work, we define an explicit datapath architecture as a class of processor architectures...
that exposes the details of the datapath, including the datapath buffers, to the software. In explicit datapath architectures, the software has more flexibility in utilizing the small but efficient datapath buffers, which can lead to a significant reduction of RF accesses, and consequently smaller RF energy consumption. However, this does not come for free. The fine-grained control capability may cause higher overhead, such as increased instruction size. Therefore, to achieve improvement in overall energy efficiency, optimizations in hardware and software for reducing the overhead are required. In this chapter, we propose architecture and code generation techniques that improve the energy-efficiency of embedded processors by leveraging the flexibility provided by an explicit datapath.

A RISC architecture based on the OpenRISC [102] is used as the baseline. The instruction set is trimmed such that only essential integer operations are supported. Table 3.1 lists the key parameters of the baseline ISA. Note that the instruction width is reduced from 32 bits to 24 bits. Similar to ARM Thumb [112] and MIPS16 [115], the more compact instruction set can lead to higher efficiency for embedded processors. The details of the baseline RISC ISA are described in Appendix A. The datapath of the baseline is similar to the one shown in Figure 2.4. More details of the baseline architecture can be found in Section 2.2.1. The RF

Figure 3.2: Power breakdown of a RISC processor running FIR (excluding memories). The reference architecture is described in Section 2.2.1 and Appendix A. The numbers are obtained by simulating a netlist synthesized with TSMC 40nm low-power library and estimating power using the library physical information. Note that the other part may contain circuit that is actually part of the RF. Because the synthesis tool often performs optimization across RTL modules, as a result it is difficult to get accurate breakdown in respect to the original modules in the RTL codes.

Table 3.1: Key parameters of the baseline RISC ISA.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction width</td>
<td>24 bits</td>
</tr>
<tr>
<td>Datapath width</td>
<td>32 bits</td>
</tr>
<tr>
<td>Pipeline stages</td>
<td>4</td>
</tr>
<tr>
<td>Register file</td>
<td>32 entries, 2R1W</td>
</tr>
<tr>
<td>Opcode</td>
<td>6 bits</td>
</tr>
<tr>
<td>Immediate</td>
<td>8 bits</td>
</tr>
</tbody>
</table>
in the baseline architecture is small and simple, and yet it consumes a considerable amount of energy. In this chapter we propose an architecture that exposes the bypassing network to the software to reduce the accesses to the RF. The proposed architecture exploits the fact that most variables in many programs are only used for a few times shortly after their values are produced. As it relies heavily on software to effectively utilize the explicit bypassing datapath, we design and implement a compiler that generates efficient code for the proposed architecture. Extensive experiments are carried out. The results show that the proposed solution is effective. On average over 70% of RF accesses are eliminated, resulting in an average reduction of 15% in the core energy, with almost no overhead in performance and code size. A 9.19% reduction in total energy consumption is observed when memory is taken into account. The proposed solution achieves a good trade-off between efficiency and control flexibility, which leads to an energy efficient processor architecture.

The remainder of this chapter proceeds as follows: Section 3.1 gives an overview of the potential to reduce register file accesses by properly utilizing the bypassing network. Different options for implementing an explicit datapath in processors are discussed in Section 3.2. And we propose an architecture that is balanced between flexibility and overhead. Section 3.3 presents code generation techniques that can efficiently utilize the proposed architecture. Extensive experiments that demonstrate the effectiveness of the proposed architecture and algorithms are shown in Section 3.4. Section 3.5 discusses the related work on explicit datapath architectures and compilation. Section 3.6 summarizes the findings in this chapter.

3.1 Reducing Register File Accesses

In a typical RISC ISA, the register file (RF) is involved as both the source and the destination location in almost every instruction. Usually, the result of each operation (a variable) in a program is assigned to a register in the RF. In general, there are three possible ways to reduce accesses to the RF:

- **Operand bypassing**: a source operand is fetched from a datapath buffer (for example, a pipeline register) instead of the RF.
- **Dead result elimination**: if all consumers of a variable use its value before it exits the datapath, it is not necessary to store it back to the RF.
- **Operand sharing**: if a source operand is already available at the input port of the function unit to execute the operation (usually because a previous operation uses the same operand), there is no need to read or move this operand.

However, in conventional architectures, it is difficult to effectively reduce RF accesses. For example, in a RISC processor, source operands are fetched in the decode stage, where RF accessing and bypassing are done simultaneously. To skip RF accesses for bypassed variables, it requires earlier check for data hazard, which needs extra logic, and may result in a longer critical path or an extra pipeline stage. Dead result elimination is particularly difficult in conventional architectures, as the liveness information of each variable is usually not available to
Table 3.2: Kernels used in the experiments.

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Description</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Histogram</td>
<td>256-bin histograming</td>
<td>Image processing</td>
</tr>
<tr>
<td>FIR</td>
<td>5-tap finite impulse response filter</td>
<td>Image processing</td>
</tr>
<tr>
<td>IDCT</td>
<td>2D 8x8 Inverse cosine transformation</td>
<td>Codec</td>
</tr>
<tr>
<td>YUV2RGB</td>
<td>YUV to RGB color space conversion</td>
<td>Image processing</td>
</tr>
<tr>
<td>MatVec</td>
<td>Matrix vector multiplication</td>
<td>Linear algebra</td>
</tr>
<tr>
<td>CRC</td>
<td>Cyclic redundancy check code calculation</td>
<td>Network/Storage</td>
</tr>
<tr>
<td>DES</td>
<td>The Data Encryption Standard algorithm</td>
<td>Security</td>
</tr>
</tbody>
</table>

Table 3.3: Statistics of kernel loops. The first row shows the number of loops in the kernels listed in Table 3.2. The second row shows the number of LLVM IR operations in the loops. The third row shows the percentage of local variables in the loops with less than 3 uses.

<table>
<thead>
<tr>
<th></th>
<th>Histogram</th>
<th>FIR</th>
<th>IDCT</th>
<th>YUV2RGB</th>
<th>MatVec</th>
<th>CRC</th>
<th>DES</th>
</tr>
</thead>
<tbody>
<tr>
<td># Loops</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Loop IR Ops.</td>
<td>100</td>
<td>28</td>
<td>333</td>
<td>44</td>
<td>36</td>
<td>79</td>
<td>802</td>
</tr>
<tr>
<td>Low use var.</td>
<td>99.00%</td>
<td>88.24%</td>
<td>96.39%</td>
<td>91.30%</td>
<td>86.36%</td>
<td>94.92%</td>
<td>93.15%</td>
</tr>
</tbody>
</table>

the instruction decoder. So to effectively utilize datapath buffers, it requires changes in both the architecture and compiler design. Explicit datapath architectures are ideal candidates for this purpose.

Intuitively, if the number of consumers of a variable is small, there is a high chance that all its consumers use the variable before it exits a pipelined datapath and get written to the RF. Figure 3.3 illustrates that in a 4-stage RISC like the baseline architecture, if a variable is consumed less than 3 times, it is possible that all its consumers fetch the value from pipeline registers. In many streaming applications, a variable is usually used for only a few times. We verify that by analyzing the 7 kernels listed in Table 3.2. Table 3.3 shows the percentage of local variables that are used less than 3 times in the most intensive loops. The numbers in Table 3.3 suggest that for many applications from various domains, the potential to efficiently utilize the datapath buffers at the top of the processor storage pyramid in Figure 3.1 is very high. Therefore it is worthwhile to exploit explicit datapath architectures to reduce RF accesses and to improve energy efficiency.

### 3.2 Architectures with Explicit Datapath

The key concept of explicit datapath architectures is that more details of the datapath, including datapath buffers like pipeline registers, are exposed to the software. The idea dates back to *microcode* that was introduced in the development of early microprocessors [143]. There are

---

1The numbers only include uses within the same basic block.
many options to implement explicit datapath. Many explicit datapath architectures use concepts similar to horizontal microcode, in which instructions specify the exact control signals for the datapath in each cycle. Unlike conventional architectures in which the microprograms are hidden and immutable\(^1\), explicit datapath architectures allow the software to have directly control over the datapath.

Having more flexibility in controlling the datapath gives the software more freedom to explore the architecture and allows the compiler to generate more efficient instruction sequences. However, the extra flexibility often comes with overhead, especially for processors that require high programmability.

In Section 3.2.1, we first look at the architecture that gives full flexibility to the software, namely, the Transport-Triggered Architecture (TTA). A TTA-based processor has to pay a price in code density for the flexibility. To achieve a design that is more balanced between flexibility and control overhead, we propose a processor that uses explicit bypassing in a RISC datapath in Section 3.2.2.

### 3.2.1 Transport-Triggered Architectures

The Transport-Triggered Architecture (TTA) proposed by Corporaal is a type of processor architecture that exposes very low level details of the datapath to software [29, 36]. The core concept in TTA is that data movement is the only thing to control in the processor datapath, and operations are a side-effect of data movement. Figure 3.4 illustrates the datapath of a TTA-based processor. A TTA processor consists of function units (FUs), register files (RFs) and data transport buses. The instructions only specify how data items are transported on these buses. An FU only performs calculations when there is an operand transported to its trigger port. The main advantages of TTA are:

- Operation results can be transported between FUs without going through the RF.
- Adding new FUs, including special function units, is relatively easy, as no ISA changes are required.

---

\(^1\)Although in many architectures the microprogram memory can be reconfigured, such capability is usually not open to user and is only used for bug fixing.
The interconnection network between RFs and FUs can be tailored for specific application-set to improve performance and/or reduce hardware cost.

In this chapter we are particularly interested in the first point, as it can be naturally used to reduce the RF accesses for short-lived variables, leading to improvements in energy efficiency.

Figure 3.5 shows an example of the code for a TTA processor as implemented in MOVE [113] and TCE [119]. To the left is a RISC program consisting of three instructions. The instructions in the middle are TTA moves that are the result of direct translation of the RISC code. Its code density is worse than a 32-bit RISC: in MOVE architecture, encoding a move takes 16 bits, which is half the size of a 32-bit RISC instruction. Therefore the result of direct translation uses up to 50% more program bits. By the proper use of explicit data transport control in the compiler, it is possible to greatly improve the code density, as shown in the TTA code to the right of Figure 3.5. However, low code density remains one of the biggest problems of TTA-based processors [36, 56, 142]. Low code density has a negative impact on energy efficiency, as fetching instructions from the instruction memory and decoding them consume a lot of energy, which may cancel the gain caused by RF. Some of the effects of low code density are shown in Section 3.4.5.

3.2.2 Explicit Bypassing in a RISC Processor

To achieve a better compromise between flexibility of explicit datapath and control overhead, we propose to use software controlled bypassing in a conventional processor datapath. Figure 3.6 shows the datapath of the proposed RISC architecture with explicit bypassing. Compared to RISC, the noticeable differences are:

- The interconnection network between RFs and FUs can be tailored for specific application-set to improve performance and/or reduce hardware cost.
**Figure 3.6:** Datapath of a 4-stage RISC processor with explicit bypassing. The FU output ports and the pipeline registers are mapped to the RF index space.

**Figure 3.7:** Example of explicit bypassing in the proposed architecture.

- The software controls whether a source operand is coming from the register file or from the bypassing network.
- Each FU has separate input registers, as a result: i) different FUs are isolated, leading to reduced circuit switching activities; ii) the output of an FU is stable as long as the value of its input registers are unchanged, creating more bypassing opportunities.

The FU output ports and pipeline registers are exposed to the software. They are mapped to the RF address space, which is illustrated in the right of Figure 3.6. By using such mapping the proposed architecture could use the same instruction format as the baseline RISC architecture. To fetch a source operand from the bypassing network, an instruction is required to specify the corresponding register index in the source operand field. For output destination, there are three options: i) write to a register (e.g., r1); ii) write to the writeback stage pipeline register, denoted as WB; iii) only write to FU output, denoted as - -.
Figure 3.7 shows an example of a code sequence executed on the proposed architecture. In the given example, none of the instruction output values needs to be stored to the RF. Note that the use of separate input registers for the LSU enables the bypassing from the \textit{load} (available in cycle 3) to the \textit{store} instruction (uses in cycle 6). Compared to RISC, the proposed architecture requires significantly fewer RF accesses, and the number of instructions remains unchanged. The choice to make each FU as a visible source can increase bypassing opportunities. Since the input registers of each FU make the output stable before a new instruction is executed, the overhead of having separate sources for each FU is relatively low. Figure 3.8 demonstrates that compared to one source per stage, separate FU sources are able to avoid more RF accesses. When the output port of each FU can be bypassed separately, one more writeback can be eliminated compared to all FUs sharing one bypassing index. Compared to TTA, the proposed architecture has more restrictions on how the software can control the datapath. While it reduces the flexibility, the control overhead is smaller. As we show later in this chapter, the proposed design leads to a more energy efficient architecture.

3.3 Code Generation for Explicit Bypassing

The proposed explicit bypassing architecture exposes more details of the datapath to the software. This requires the compiler to adapt to the fine-grained datapath control. In the meantime, it allows the compiler to perform optimizations that would not be possible on a conventional processor, leading to more efficient code.

This section describes the compiler back-end design for the proposed explicit bypassing architecture, which is based on a compiler for a conventional RISC processor. The most important modification is related to instruction scheduling, which determines whether the explicit datapath is efficiently utilized. Figure 3.9 shows the main processing passes for compiling a function in the compiler back-end for the proposed explicit bypassing architecture. Basic control-flow (e.g., natural loops and domination-trees) and data-flow information (e.g., variable liveness) are maintained during the execution of these passes. When there are multiple functions, a function is only processed when all functions it calls except itself are compiled\footnote{In this work we assume that there are no circular calls.}. The remainder of this section shows the details of the compiler back-end. The bypass-aware instruction scheduler is described in Section 3.3.1. Section 3.3.2 discusses the issues introduced by explicit bypassing and how to handle them in the compiler. Section 3.3.3 presents long immediate optimization for
the proposed architecture. Last but not least, the register allocation is described in Section 3.3.4.

### 3.3.1 Bypass-Aware Instruction Scheduling

In this chapter, instruction scheduling is done at the basic block level. A basic block of a program is a sequence of instructions that contains only one entry point (the first instruction) and only one exit point (the last instruction). An instruction in a basic block always executes before all instructions in later positions within the same block. A basic block can be represented by a data-flow graph (DFG) $G(V,E,\delta)$, where:

- $V$ is a set of nodes. Each node in $V$ represents either an actual operation or a live-in variable.
- $E$ is a set of directed edges. An edge $e = (u,v)$ represents that node $v$ depends on $u$, where $u,v \in V$. $e$ can be either a true data dependency or false dependency.
- $\delta : E \rightarrow \mathbb{Z}$ specifies the latency of the edges. $\delta(e)$ is the latency of edge $e$ in cycles.

A schedule of a DFG $G$ is a function $\sigma : V \rightarrow \mathbb{Z}^+$ that defines the issue time for each operation in $V$. A valid schedule should satisfy the timing requirements imposed by each edge:

$$\forall e = (u,v) \in E, \sigma(v) - \sigma(u) \geq \delta(e)$$  \hspace{1cm} (3.1)

In this work, the goal of the scheduling algorithm is to find a schedule $\sigma$ for the proposed architecture that minimize the overall energy consumption, under the constraints specified by Equation 3.1.

In the proposed architecture, the register pressure in many kernels are high due to two reasons: $i)$ part of the RF index space is used by the bypassing source, resulting in a smaller RF compared to the baseline RISC; $ii)$ because only 8-bit immediate value can be encoded in an instruction, more registers may be needed to hold the long immediate values (see Section 3.3.3). Prior to scheduling, the DFG is partitioned into sub-graphs using Algorithm 3.1 to make the
bypassing easier and keep the register pressure low. The algorithm starts by putting nodes with not out-going edges in a queue. In each iteration, the head of the queue is picked and a new partition is created for this node. The new partition is grown by recursively adding nodes that contains consumers only in this partition. Each partition $p$ can be represented by a tree structure, in which all nodes except the root are used only within $p$. The fact that nodes in $p$ are consumed locally is helpful in the proposed explicit bypassing architecture: if nodes in $p$ are scheduled close to each other, few or even no registers are required to store the intermediate results, such as the example in Figure 3.10. To maximize the bypassing, nodes in $p$ are numbered by running the Sethi-Ullman algorithm on the tree that represents $p$ [136]. The numbers assigned to the nodes represent the scheduling order which minimizes the number of registers required to compute the tree.

The scheduler in this work schedules a basic block in a bottom-up fashion, that is, a node $v$ in $G$ is scheduled after all nodes that depend on it are scheduled. There are two main advantages in using bottom-up scheduling: $i$) branch delay slots can be easily handled by first scheduling the branch instruction in the second to last cycle, as it is always ready to be scheduled in bottom-up scheduling; $ii$) the scheduler knows precisely whether the result of the instruction to be scheduled needs to be written back to the register file. Algorithm 3.2 depicts the scheduling algorithm used in this work. When selecting the next instruction (line 11), the schedule tries to keep nodes in the same partition close to each other by first choosing the nodes from the partition with minimum $\text{PartitionOrder}$. And within a partition, nodes are ordered by their Sethi-Ullman numbers. If there is no ready instructions from the same partition, the scheduler can choose one from another partition to fill the empty slot.

### 3.3.2 Handling Explicit Bypassing

After instruction scheduling, the compiler scans through the scheduled instructions in each basic-block. During the scan, the compiler:

- Checks if a source operand should be fetched from the bypassing network. If so, it sets the register index to the corresponding value. This is necessary because there is no automatic bypassing in the proposed architecture.
Algorithm 3.1: Data flow graph partitioning

**Input:** Data flow graph $G(V,E)$

**Output:** Partition $P$

1. $P \leftarrow \{\}$
2. $roots \leftarrow \{\}$ // Stack for root nodes
3. // Find initial roots
4. **foreach** $v \in V$ **do**
5.   **if** $v$ has no out-going edge **then**
6.     $roots$.push($v$)
7. **end**
8. **end**
9. $processed \leftarrow \emptyset$
10. **while** roots is not empty **do**
11.   $r \leftarrow roots$.pop()
12.   $p \leftarrow \{\}$ // A new partition
13.   $nodes \leftarrow \{r\}$ // A FIFO queue for nodes of $p$
14.   // Grow the partition as much as possible
15.   **while** nodes is not empty **do**
16.     $n \leftarrow nodes$.pop_front() // Dequeue from node FIFO
17.     $p$.add($n$)
18.     $processed$.add($n$)
19.     **foreach** $n$’s producer $u$ **do**
20.       **if** $u \notin processed$ and $u$ has no consumer outside $p$ **then**
21.         $nodes$.push_back($u$) // Add $u$ to node FIFO
22.       **end**
23.   **end**
24.   $P$.add($p$)
25.   // Check unprocessed nodes
26. **foreach** $u \in V, u \notin processed$ **do**
27.       **if** $u$’s consumers are processed and $u \notin roots$ **then**
28.         $roots$.push($u$)
29.       **end**
30. **end**
Algorithm 3.2: Basic-Block bottom-up scheduling

Input: A basic block $B$, its data flow graph $G(V, E)$ and its partition $P$
Output: A valid schedule $\sigma$ for $G$

// The actual order is the reverse of scheduled order
// Handle branch delay slot by scheduling the branch first

1 if $B$ has a branch $j$ then
   \[ \sigma[j] \leftarrow 1 \]
end

4 foreach $p \in P$ do partition order initialization
   \[ O[p] \leftarrow \infty \]
end

7 $t \leftarrow 0$
8 $o \leftarrow 0$

9 while Not all nodes in $G$ is scheduled do
   \hspace{1em} while Resource available in $t$ do
      \hspace{2em} $v \leftarrow$ eligible node with minimum lexical order of ($BypassingConflict$, $PartitionOrder$, $NodePartitionNum$, -$Bypassing$)
      \hspace{2em} if No node is selected then
      \hspace{3em} break
      \hspace{2em} end
      \hspace{2em} $\sigma[v] \leftarrow t$
      \hspace{2em} $p \leftarrow P[v]$
      \hspace{2em} if $O[p] == \infty$ then
      \hspace{3em} $O[p] \leftarrow o$
      \hspace{3em} $o \leftarrow o + 1$
      \hspace{2em} end
      \hspace{2em} Update readiness of predecessors of $v$
      \hspace{2em} Update resource usage
   \hspace{1em} $t \leftarrow t + 1$
end
- If all uses of a non live-out value are bypassed, mark the value as non-writeback (to FU output port or writeback stage register).

However, this scan only sets the instruction bypassing states within a basic block. The remainder of this subsection explains how the compiler handles explicit bypassing across basic blocks and function boundaries.

**Control-Flow Join-Point Issues**

In the proposed architecture, all bypassing in the datapath is explicitly controlled by software, which means each operand has to come from a unique location, either a register in the RF or a bypassing source. At control-flow join-points where operand values are computed in different blocks, it is possible that a source operand is available at different locations depending on which control-flow path the program actually takes, resulting in a bypassing conflict. An example of join-point a bypassing conflict is illustrated in Figure 3.11. In the example, the first source operand of the store instruction (SW) in block $F$ may come from block $A$ or $B$, in which the value of $r9$ is computed in different FUs. To generate correct code, the compiler has to avoid such conflicts.

In this work, the bypassing conflict problem is resolved by ensuring that the join-point consumer (e.g., the store instruction in Figure 3.11) could fetch the value from the RF. During instruction scheduling, the compiler avoids choosing instructions that could potentially cause join-point issues. As described in Section 3.3.1, the scheduler schedules a basic block in bottom-up fashion. To minimize join-point issues, the scheduler avoids scheduling instructions that may cause bypassing conflicts at the end of a block when it is possible. If all successor blocks of the current block are scheduled, the scheduler can determine exactly which instructions can cause bypassing conflicts and avoid choosing them if there are alternatives ($BypassingConflict$ in line 11 of Algorithm 3.2). When a successor block is not scheduled yet, the earliest possible time (i.e., the topological order) is used for estimation. To increase the accuracy, the compiler only schedules a block when all its successor blocks in the control-flow graph are scheduled, or when it is the last unscheduled block in the original program order.

After instruction scheduling and setting the bypass state within each basic block, the compiler checks for bypassing conflicts at basic block boundaries. The detected conflicts are eliminated by padding NOPs to ensure the consumer can access the operand value in the RF. The compiler pads NOPs according to the following two rules:
• At call-sites, padding is done in the caller. Padding in the caller avoids unnecessary padding when there are multiple callers and only some of them have bypassing conflicts.

• At join-points within a function, padding is done either in the producer blocks (A and B in Figure 3.11) or in the consumer block (F in Figure 3.11). The padding location is chosen based on the padding cost. The padding cost for a block is calculated by a loop-based estimation \(10^{\text{loop
depth}_B}\), in which \(\text{loop
depth}_B\) is the depth of the loop \(B\) is in (0 if \(B\) is not in a loop). By using this cost estimation the compiler performs padding in blocks that have lower performance overhead.

### 3.3.3 Long Immediate Optimization

In the 24-bit ISAs of both the baseline RISC architecture and the proposed explicit bypassing architecture, there are only 8 bits for encoding immediate values in the instruction. As a result, only small integers can be directly encoded in an instruction: \(-128 \sim 127\) for 2’s complement signed integers or \(0 \sim 255\) for unsigned integers. To use an integer immediate outside the 8-bit range, an extra immediate instruction that loads the higher bits of the integer is required, resulting in performance and energy overhead.

In this work, the compiler optimizes for long immediate integers by promoting them to registers when it is beneficial. If a long immediate is promoted, it is stored in a program-wide constant pool in the data memory. The value of a long immediate is loaded to a register before it is used, i.e. the loading should dominate all uses of the immediate. To reduce the runtime cost, the loading is done in the first non-loop common dominator block of all the basic blocks that use it. If a long immediate is used in a loop, or is used more than once, it is beneficial to promote it to a register. In the example shown in Figure 3.12, the immediate 512 is used by two instructions in a loop, resulting in two simm instructions that load the higher bits. By promoting the immediate to register \(r10\), both simm can be removed, while the loading of the value is done outside the loop. As a result, the number of instructions in the loop is reduced, leading to performance and energy efficiency improvement. The optimization described in this sub-section can reduce the cost of using long immediate when there are sufficient number of registers. However it may not be effective when the register pressure is already high, in which case the long immediate handling is still an open issue.

### 3.3.4 Register Allocation

After the bypassing states are set, the values that need to be assigned to physical registers are known. The compiler uses a graph-coloring based algorithm [20] to allocate registers for such values for each function. Note that when estimating the spilling cost, a register that holds a long immediate (see Section 3.3.3) is considered to have lower spilling cost compared to a normal variable, because its value can be loaded by a combination of immediate instruction instead of being stored to and loaded from the stack. When the instruction schedule is changed due to spilling, the compiler needs to re-run the bypassing state setting processes described in Section 3.3.2.
Figure 3.12: Example of promoting a long immediate to a register. A \textit{simm} instruction loads the higher bits for the next instruction that uses a long immediate. In the original code, the immediate value loaded is \((2^{<<8})|255 = 512\).

Register Usage at Call-Sites
When there is a function call, the caller is responsible for saving and restoring the live-variables in the registers that are used by the callee. In this work, the compiler has complete information of all functions at compile time. A function is only compiled after all its callee functions are processed. At a call-site of a direct call, the compiler is able to save only the live registers that are used by the callee and by functions called by the callee. For a callee function on the other hand, it is more difficult to calculate the registers that need to be saved when the function is called at multiple locations. Therefore it is more efficient to let the caller save the registers across a call-site. In the work of this chapter, we do not handle indirect calls, but they can be handled with the same scheme. For an indirect call, the caller need to save all the live registers that are used in any of the potential callees. If the set of potential callees are unknown, the caller needs to save all the live registers.

To reduce the number of registers that need to be saved across function calls, the compiler avoids using physical registers that are used by the callees of the current function. This is done by sorting the list of available physical registers according to the register usage of the callees.

3.4 Experimental Results
To evaluate the proposed architecture and algorithms, extensive experiments are carried out on realistic platforms. The experimental setup is described in Section 3.4.1. Section 3.4.2 gives the silicon area results of the proposed architecture. The performance results and energy results are presented in Section 3.4.3 and Section 3.4.4, respectively. In Section 3.4.5, the proposed architecture is compared against another explicit datapath architecture, a TTA processor built with the TCE framework.
Table 3.4: Parameters of the proposed processor used in the experiments

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction width</td>
<td>24 bits</td>
</tr>
<tr>
<td>Datapath width</td>
<td>32 bits</td>
</tr>
<tr>
<td>Pipeline stages</td>
<td>4</td>
</tr>
<tr>
<td>Register file</td>
<td>28 entries, 2R1W</td>
</tr>
<tr>
<td>Opcode</td>
<td>6 bits</td>
</tr>
<tr>
<td>Immediate</td>
<td>8 bits</td>
</tr>
<tr>
<td>Instruction memory</td>
<td>12 KB (24-bit)</td>
</tr>
<tr>
<td>Data memory</td>
<td>16 KB (32-bit)</td>
</tr>
</tbody>
</table>

3.4.1 Experimental Setup

Two processors are used in the experiments in this section, namely, the RISC baseline processor and the proposed explicit bypassing processor. The parameters of the ISA of the processor used in the experiments in this section are listed in Table 3.4. To estimate energy consumption, both processors are implemented in synthesizable Verilog RTL and synthesized to gate-level netlist with TSMC 40nm low-power library targeting 100MHz. Clock gating is used to minimize dynamic power consumption.

In the experiments, 7 kernels from different applications are used. The kernels are listed in Table 3.2. For both the baseline architecture and the proposed architecture, all kernels are written in C and are compiled using the compiler described in Section 3.3 with maximal optimization level (-O3). The core\textsuperscript{1} energy consumption is estimated with the back-end information and actual toggle rate obtained from gate-level simulation. The energy consumption of the memory is estimated using CACTI [114]. The results of the TTA processor are obtained using the compiler and simulator in the TCE framework.

3.4.2 Area

The two processors have similar area and timing. Table 3.5 shows the area of the baseline processor and the proposed processor, including absolute numbers reported by the synthesis tools and relative values. Both processors are synthesized with TSMC 40nm low-power library. The processor with explicit datapath is smaller mainly due to the following reasons: \textit{i}) the physical register file is smaller (28 entries vs. 32 entries); \textit{ii}) the logic for automatically detecting bypassing and selecting operand sources is removed.

3.4.3 Performance

Table 3.6 shows the relative cycle overhead of the proposed architecture, which is caused by \textit{i}) extra NOPs inserted for resolving the control-flow join-point problems described in Sec-

\textsuperscript{1}In this work core is defined as all parts of the processor except the data memory and instruction memory.
Table 3.5: Core area reported by synthesis tools. The numbers inside parentheses are relative values. The unit of absolute values $\mu m^2$. The Cell area is the area occupied by logical cells, which is calculated using the technology library. The Net area is the estimated using the physical library information.

<table>
<thead>
<tr>
<th></th>
<th>Cell</th>
<th>Net</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>10806 (1.000)</td>
<td>23411 (1.000)</td>
<td>34216 (1.000)</td>
</tr>
<tr>
<td>Bypass</td>
<td>10241 (0.948)</td>
<td>22809 (0.974)</td>
<td>33050 (0.966)</td>
</tr>
</tbody>
</table>

Table 3.6: Cycle overhead of explicit bypassing.

<table>
<thead>
<tr>
<th></th>
<th>Histogram</th>
<th>FIR</th>
<th>IDCT</th>
<th>YUV2RGB</th>
<th>MatVec</th>
<th>CRC</th>
<th>DES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overhead</td>
<td>0.07%</td>
<td>0.01%</td>
<td>1.13%</td>
<td>0.01%</td>
<td>0.00%</td>
<td>0.40%</td>
<td>0.36%</td>
</tr>
</tbody>
</table>

3.4.4 Energy Consumption

As depicted in Figure 3.13, the number of register file (RF) accesses is reduced dramatically. On average, 72.7% of the RF accesses are eliminated. The maximal reduction happens in the CRC case, where 86.4% of register accesses is bypassed. Moreover, the number of register writes is reduced by 86.0%, which is more than the reduction of the number of register read (64.6%). From the energy standpoint, eliminating register writes is more important as a register write tends to consume more energy than a register read [10].

The dramatic reduction in register accesses leads to substantial energy saving in the core. Figure 3.14 shows the normalized core energy consumption. Comparing to the baseline processor, the proposed architecture reaches a maximal core energy reduction of 23.9% in the CRC case, and of 15.0% on average.

As presented in Section 3.4.3, the proposed architecture has little overhead in cycles. Therefore, there is also little overhead in memory energy consumption. Figure 3.15 shows the normalized total energy consumption. The memory energy consumption estimated by CACTI is listed in Table 3.7. The total energy consumption is reduced by 17.7% in the CRC kernel. On average, the total energy reduction is 9.19%.

---

1Core is defined as the control-path and datapath of a processor. A complete processor system also includes instruction memory and data memory, which are designed with tools and technologies different from the core part. The total energy consumption includes both the core energy consumption and the memory energy consumption.
Figure 3.13: Normalized number of RF accesses.

Figure 3.14: Normalized core energy consumption.

Figure 3.15: Normalized total energy consumption.
Table 3.7: Estimated energy consumption of the memories used in the experiments.

<table>
<thead>
<tr>
<th>Memory</th>
<th>16kB 32-bit</th>
<th>12kB 24-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Energy per access (pJ)</td>
<td>2.92</td>
<td>2.49</td>
</tr>
</tbody>
</table>

Figure 3.16: The connection between RF and FUs in the 2-issue TTA processor used in the experiments.

3.4.5 Comparison with TCE

To compare the energy efficiency with an architecture with an explicit datapath in a different style, we use the TCE [119] framework (Version 1.8) to build TTA processors to run the same benchmark applications as the proposed architecture.

In this chapter, two processors are built with the TCE framework: a 2-issue TTA processor and a 3-issue TTA processor, which are referred to as TCE-2 and TCE-3 in the remainder of this section, respectively. The datapath of the TCE-2 is shown in Figure 3.16. The immediate unit (IMM) is added in TCE-2, which is a read-only register file for storing long immediates (that cannot be encoded in move slots directly). The content of the immediate unit is configured by the instruction buses. Though TCE-2 has the same type and number of FUs as the baseline RISC processor, the load-store unit (LSU) in the TCE framework does not have the same address calculation capability (base+offset), which may have negative impact on the results. Therefore a 3-issue TTA processor TCE-3 is used in the experiments in this section, which contains a separate adder and a partially connected bus for address calculation. Figure 3.17 depicts the datapath of the TCE-3. The key parameters of the TTA processors used in the experiments are listed in Table 3.8. A key observation is that the TTA instructions are much wider (41-bit and 57-bit vs. 24-bit) than the baseline. The benchmark kernels are compiled by the tcecc compiler with maximum optimization level (-O3), and with the software bypassing algorithm described.

Figure 3.17: The connection between RF and FUs in the 3-issue TTA processor used in the experiments. Compared to the 2-issue processor in 3.16, an adder and a partially connected bus are added.
Table 3.8: Parameters of the two TTA processors built with TCE

<table>
<thead>
<tr>
<th>Parameter</th>
<th>TCE-2</th>
<th>TCE-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Move Slots</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>FU</td>
<td>1 ALU, 1 MUL, 1 LSU</td>
<td>1 ALU, 1 ADD, 1 MUL, 1 LSU</td>
</tr>
<tr>
<td>Integer RF</td>
<td>$32b \times 32, 2R1W$</td>
<td>$32b \times 4, 1R$</td>
</tr>
<tr>
<td>Immediate Unit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Boolean RF</td>
<td>$1b \times 2, 1R1W$</td>
<td></td>
</tr>
<tr>
<td>Instruction Width</td>
<td>41 bits</td>
<td>57 bits</td>
</tr>
</tbody>
</table>

Figure 3.18: Instruction format of TCE-3. The instruction format of TCE-2 is the same as TCE-3, except that there is no Move Slot 3. The source field of each slot can encode an 8-bit short immediate.

Figure 3.19 shows the comparison of normalized cycle count between the proposed architecture, TCE-2 and TCE-3. On average, the TCE-2 uses 50.4% more cycles than the proposed architecture, and the TCE-3 uses 5.9% more cycles than the proposed architecture. This implies that the instruction fetch energy of TCE-2 and TCE-3 is much higher, as the instructions of the TCE-2 and the TCE-3 are much wider than the proposed architecture.

Figure 3.20 depicts the comparison of RF accesses between the proposed architecture and the two TCE-based processors. On average, the TCE-2 has 54.5% more RF accesses than the proposed architecture, and the TCE-3 has 45.7% more RF accesses. The breakdown of reads and writes shows that the TCE-2 has 178% more RF writes, and the TCE-3 has 165% more RF writes. As the energy consumption of an RF write is higher than a read, these numbers indicate much more energy consumption in the RF.

Effect of Code Compression in TCE

The TCE framework supports dictionary-based code compression [58]. Table 3.9 shows the result of applying instruction dictionary compression for the kernels used in the experiments on the TCE-3 processor. On average, the dictionary-based code compression reduces the code
**Figure 3.19:** Cycle count of the proposed architecture and TCE (normalized to baseline).

**Figure 3.20:** RF accesses of the proposed architecture and TCE (normalized to baseline).
### Table 3.9: Result of code compression for TCE-3.

<table>
<thead>
<tr>
<th></th>
<th>Original Size (Bytes)</th>
<th>Compressed Size (Bytes)</th>
<th>Dictionary Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Histogram</td>
<td>4005 (57b × 562)</td>
<td>633 (9b × 562)</td>
<td>57b × 265</td>
</tr>
<tr>
<td>FIR</td>
<td>2907 (57b × 408)</td>
<td>408 (8b × 408)</td>
<td>57b × 220</td>
</tr>
<tr>
<td>IDCT</td>
<td>4881 (57b × 685)</td>
<td>771 (9b × 685)</td>
<td>57b × 465</td>
</tr>
<tr>
<td>YUV2RGB</td>
<td>3221 (57b × 452)</td>
<td>452 (8b × 452)</td>
<td>57b × 253</td>
</tr>
<tr>
<td>MatVec</td>
<td>3150 (57b × 442)</td>
<td>442 (8b × 442)</td>
<td>57b × 246</td>
</tr>
<tr>
<td>CRC</td>
<td>3620 (57b × 508)</td>
<td>572 (9b × 508)</td>
<td>57b × 257</td>
</tr>
<tr>
<td>DES</td>
<td>8536 (57b × 1198)</td>
<td>1348 (9b × 1198)</td>
<td>57b × 479</td>
</tr>
</tbody>
</table>

### Table 3.10: Energy consumption of memories with different width and size.

<table>
<thead>
<tr>
<th>Entries</th>
<th>512</th>
<th>1024</th>
<th>2048</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-bit</td>
<td>0.39 pJ</td>
<td>0.71 pJ</td>
<td>1.35 pJ</td>
</tr>
</tbody>
</table>

size by 32.4% (ratio between (compressed size + dictionary size) and uncompressed size). Table 3.10 depicts the energy consumption of the memories that can be used for TCE-3 instruction memory and dictionary\(^1\). Although code compression is able to dramatically reduce the code size, TCE-3 still suffers from bad code density. In particular, even though the width of a compressed instruction is considerably smaller than the original one, the dictionary used for decoding is still 57 bits wide. Using a combination of 2048-entry 8-bit memory (for instructions) and 512-entry 56-bit memory (for dictionary), the energy consumption is 3.1 pJ per instruction, which is 24.5% more than the 4096-entry 24-bit instruction used in the proposed architecture. Similar result is also observed when applying code compression to the TCE-2 processor. The decompression hardware also introduces additional overheads in area and energy. In addition, code compression can also be applied to architectures other than TTA when it is appropriate.

In summary, the proposed architecture and compiler design is able to dramatically reduce the number of register file accesses with little overhead. As a result, the energy efficiency of the processor is greatly improved.

### 3.5 Related Work

In microprocessors, the register file is one of the central components, which accounts for considerable amount of energy. Zyuban and Kogge give a detailed analysis of RF power consumption in [168]. In [128], Rixner et al. present the trade-offs in different register file designs for media processors.

\(^1\)Due to the limitation of CACTI, the energy consumption of 9-bit and 57-bit memories cannot be estimated. The numbers for 8-bit and 56-bit wide memories are used as approximation.
The idea of exposing part of the bypass network to the software to reduce RF accesses has
been exploit in a few previous studies. Lozano and Gao propose a scheme to use the reorder
buffer (ROB) to dynamically eliminate register file accesses for short-lived variables in a su-
perscalar processor [93]. In [93], the proposed architecture exposes the ROB to the software
and the compiler has to tag the short-lived variables and allocate so-called symbolic registers
for them. Tseng and Asanovic propose several techniques at circuit and architecture level to
reduce the RF access energy [150]. Sami et al. propose a design approach that exploits by-
passing of short-lived variables in the datapath of VLIW processors [133, 134]. Compared to
their work, the method in chapter differs in the following aspects: i) bypassing information is
encoded without using extra bits; ii) the compiler is aware of the explicit bypassing and opti-
mizes code for it. Balfour et al. introduce the ELM architecture that reduces RF accesses by
using a combination of software bypassing and hierarchical RFs [10, 11]. However in ELM the
hardware bypassing logic is retained for code density consideration. In this work, we show that
in the most frequently executed region of typical streaming applications, the use of pure soft-
ware bypassing does not have negative impact on code density. Finlayson et al. propose a static
pipelining architecture that exposes internal registers in the instruction set architecture [37].
Compared to the proposed architecture, the static pipeline architecture requires more extensive
changes in the processor architecture and the compiler [38]. In [161], Woh et al. address similar
problems in SIMD context. RF partitioning is used as the solution, in which a 4-entry RF is
used for short-lived variables instead of the 16-entry main RF. The TTA is proposed by Cor-
poraal [29]. The MOVE [113] and the TTA-based codesign environment (TCE) [119] are the
two most well known implementations of TTA. He et al. propose MOVE-Pro that addresses the
code density issue for TTA processors [56].

A compiler is known to play a key role in reducing RF energy consumption even on con-
tventional architectures. Mehta et al. propose register relabeling in the compiler that reduces
the switching cost in the RF [95]. For an architecture with explicit datapath, the compiler is
the key to achieve high efficiency, as more details of the datapath are exposed to the software
compared to conventional architectures. The compiler for ELM uses a graph-base scheduler
to schedule instructions for the explicit datapath [106]. Hoogerbrugge [62] and Janssen [69]
analyze different aspects of compiler back-end design for TTA. Guzma et al. discuss the per-
formance impact of software bypassing in TTA [52], and the power consumption implication
is in [51, 53]. Reshadi and Gajski present a compiler design for No-Instruction-Set-Computing
(NISC) architecture [126]. NISC resembles TTA, but its control overhead is less of a concern
as it mainly targets platforms that are able to reconfigure the control logic. She et al. propose a
compiler design for MOVE-Pro that overcomes problems causing inefficiency in conventional
TTAs [142]. Comprehensive comparison between TTA and RISC/VLIW counterparts is pre-
sented in [56] and [142], which shows that a 2-issue MOVE-Pro is able to achieve similar code
as a 32-bit RISC. However, its code density is still low compared to architectures with com-
 pact ISA (16-bit or 24-bit). The control flexibility in architecture like TTA and NISC makes
them ideal candidates for implementing application-specific accelerator, but the relatively high
control cost often leads to high overhead where programmability requirement is high. In this
chapter, we propose an algorithm that partitions the DFG prior to running a bottom-up list scheduler. The proposed DFG partitioning algorithm is similar to the one used in by Govindarajan et al. [48]. The main difference is that to better utilize the explicit bypassing feature, the proposed algorithm partitions the DFG into tree-like sub-graph instead of linkages in [48].

In this chapter, we proposed a processor architecture that requires relatively small modification to conventional RISC architecture. Furthermore, we presented the compiler design for this architecture. Compared to TTAs, the proposed architecture is more suitable for designing an embedded processor that is capable of running programs in different domains efficiently.

3.6 Summary
In many programs, most of the variables are only used locally for a few times. Conventional processors store these variables in the register file (RF), resulting in a power-hungry RF that consumes a considerable portion of the core energy. Our experiment shows that in an FIR filter running on a RISC processor, the RF consumes over 18% of the core energy. In this chapter, we explored the use of explicit datapath to improve energy efficiency of processors by reducing accesses to the register file (RF). By having fine-grained control, architectures with explicit datapath can eliminate unnecessary temporary storage of intermediate results. However, the fine-grained control comes with overhead. In particular, many architectures with explicit datapath suffer from poor code-density as much more control information need to be encoded into the instructions compared to conventional processor architectures.

To achieve a balance between control granularity and overhead, we proposed a processor architecture with software-control bypassing. The proposed architecture only requires small modification to conventional RISC architecture. The design of a compiler for the proposed architecture is presented. A bottom-up bypass-aware scheduling algorithm is proposed for effectively utilizing the explicit datapath. Extensive experiments show that the proposed solution is able to achieve high energy efficiency. Over 70% of RF accesses are eliminated. As a result, the core energy is reduced by 15%. There is almost no overhead in performance and code size, a 9.19% reduction in total energy consumption is observed when memory is taken into account. Compared to another type of architecture with an explicit datapath, TCE, the proposed architecture and compiler are able to reduce RF accesses with much lower overhead, which leads to higher energy efficiency.
Energy Efficient Flexible Special Instructions

In many applications, complex operation patterns that contain multiple operations occur frequently. Figure 4.1 shows some examples of such patterns. Synthesizing hardware to execute special instructions that support such patterns has become a common approach in application specific instruction-set processor (ASIP) design [24, 76, 90]. Special instructions are able to dramatically reduce the number of instructions and the amount of communication between data-path components, which have great impact on performance as well as energy consumption. The overhead of supporting special instructions in ASIPs is relatively low as the instruction set architecture (ISA) can be tailored for a specific set of applications. Commercial products that adopt this approach can be found on the market, for example, Tensilica Xtensa [120], Stretch software configurable processors [44, 118], and IMEC ADRES reconfigurable processor [96]. The successful uses of special instructions in ASIPs have shown that it is an effective approach towards improving energy efficiency.

However, supporting special instructions in generic embedded processors to achieve high energy efficiency is still a challenging task. To be efficient in general purpose computing, it is intuitive to support many complex operation patterns, so that applications from different domains can benefit from it. However, in most mainstream general purpose processor archi-

![Figure 4.1: Examples of special operations.](image-url)
tectures, only a few complex instructions are used\(^1\), because supporting arbitrary operation patterns incurs large overhead that might diminish the gain of using these patterns. Many optimization methods in ASIPs are not applicable because the processor has to support a much wider range of applications. From energy efficiency perspective, the overhead includes:

- More instruction bits are needed to encode opcodes for all possible patterns and extra operands in the special instructions. In a compact ISA like the ARM Thumb [112], the problem is even more severe as the instruction width is very small. Making the instruction wider results in higher energy consumption in instruction fetching.

- The register file (RF) needs more ports to provide sufficient bandwidth for the special function units. According to the model in [146], the power and area of RF have quadratic dependency on the number of ports, and the delay has linear dependency on the number of ports, indicating that the cost of adding extra ports is high.

- The complexity in many parts of the processor is increased, including the instruction decoder and bypassing network. Both timing and energy consumption are affected.

More importantly, the overhead affects not only the special operations, but also the normal instructions that do not need the added resources. Therefore, to achieve high energy efficiency, the support for special instructions needs to have low overhead, while still being able to support applications from different domains. In this chapter, we tackle the problem of integrating flexible special instruction support in a generic embedded processor with a compact ISA. Most previous works focused on improving the performance [24, 26, 75]. However, our primary goal is to achieve high energy efficiency. We propose a scheme for integrating a Special Function Unit (SFU) into a RISC embedded processor with an ISA that has 24-bit instruction width. The design of an SFU that supports flexible operation pair patterns is presented. To efficiently integrate the SFU into a processor with compact ISA, the proposed scheme provides good coverage for different applications with relatively low overhead by using:

- A partially reconfigurable instruction decoder that allows low-cost runtime reconfiguration to use different patterns for specific parts of applications.

- A software controlled bypass network based on the one introduced in Chapter 3, which increases operand bandwidth without extra RF ports or wider instructions.

Both the reconfigurable decoder and the explicit bypass network rely on compiler support. We design a compiler back-end that is able to generate efficient code for the proposed architecture. For each function, the back-end selects operations patterns, schedules instructions and inserts reconfiguration codes. Extensive and detailed experiments are performed to evaluate the proposed solution. The results show that the proposed solution reduces total processor energy consumption by an average of 15.8% compared to a RISC processor. Compared to an

\(^1\)In some ISAs, different addressing modes contain relative complex address calculation patterns. However, supporting complex patterns for generic computation are not common, especially for ISAs based on the RISC concept.
architecture that integrates the same SFU without the proposed reconfigurable decoder and explicit bypass network, the proposed architecture has more constraints on the operations patterns. However, it achieves almost the same reduction in cycle count with much less overhead. By introducing multi-cycle operations in the SFU to compensate for the increased critical path delay, the proposed architecture is able to improve the performance by 13.8%, with 13.1% less energy compared to the RISC processor. All in all, the proposed solution provides a good balance point between performance and energy efficiency for embedded processors with compact ISAs.

The remainder of this chapter is organized as follows. In Section 4.1, we define the type of instruction patterns that are used in this work and analyze the characteristics of these patterns in a set of applications. The design of the proposed SFU and how it is integrated into a compact RISC processor are described in Section 4.2. Section 4.3 describes the design of compiler back-end for the proposed architecture. Detailed and comprehensive experimental results are given in Section 4.4. Section 4.5 discusses the related work. Lastly, Section 4.6 concludes the findings in this chapter.

### 4.1 Special Operation Patterns

A program can be divided into a set of basic blocks, in which the dependency between operations can be represented by a data-flow graph (DFG) $G(V, E_d, E_f)$, where:

- $V$ is a set of nodes. Each node in $V$ represents either an actual operation or a live-in variable (stored in register file or encoded in instruction immediate field). Such operations are defined as basic operations.
- $E_d$ is a set of directed edges. An edge $e = (u, v) \in E_d$ represents that node $v$ consumes the value of $u$, i.e., there is true data dependency between $u$ and $v$, where $u, v \in V$.
- $E_f$ is a set of directed edges that represent false dependencies. If there exists an edge $e = (u, v) \in E_f$, $v$ cannot be executed before $u$, though $v$ does not use the output of $u$.

The DFG for a basic block is a directed acyclic graph (DAG). In this chapter, we define basic operations as operations that can be directly translated to an instruction in the RISC ISA described in Appendix A. A special operation pattern is defined as a sub-graph of a DFG that contains multiple basic operations. Figure 4.1 depicts some examples of such patterns. A Special Function Unit (SFU) is a function unit in processor datapath that directly executes one or more special operation patterns. An instruction that uses a SFU is called a special instruction. Compared to a sequence of basic instructions that perform the same computation, using a special instruction has a number of advantages:

- Fewer instructions are needed to execute the operations, resulting in less overhead in control, including instruction fetching and decoding.
- As fewer instructions are used for the same computation, the performance can be improved if the instruction throughput remains the same.
• The communication between the basic operations can be done within the FU, which is usually much more efficient.

For a certain application, some special operation patterns appear frequently [4]. In application specific instruction-set processor (ASIP) design, a common approach for improving performance as well as energy efficiency is to synthesize special function units (SFUs) that support these patterns [24, 76, 90, 166]. Different from the work of ASIP design, the goal of this work is to support special operation patterns in a RISC-like generic processor, without introducing heavy modifications to existing architecture and code generation framework. Instead of trying to support arbitrary patterns, we focus on a specific type of patterns, namely, operation pairs. In the remainder of this section, we explain the definition of operation pair patterns and the motivation of choosing this type of patterns.

4.1.1 Operation Pair Patterns

In this chapter, we want to integrate the support for special instructions without making major modification to the original RISC architecture described in Appendix A. A single-issue RISC processor typically has a register file (RF) with two read ports and one write port (2R1W). Therefore it is difficult to supply more than two source operands to a function unit (FU) in one cycle. The same holds for the destination operand. Also a RISC architecture usually uses a three-address instruction format in which only two source operands and one destination operand can be specified. In addition, the number of arbitrary operation patterns in different applications is huge. An FU that supports all these patterns not only makes the hardware very complex and inefficient, but also dramatically increases the difficulty of code generation. So in this work, we focus on operation pair patterns, i.e., patterns with two operations $a$ and $b$ that meet the following constraints:

• There is true dependency between $a$ and $b$, i.e., $(a, b) \in E_d$.

• There are at most three input operands. More formally, for a set of edges $P$ that contains all edges to $a$ or $b$ in $E_d$ except $(a, b)$, we have $|P| \leq 3$;

• At most one of $a$ and $b$ has consumers outside the pattern, i.e., at least one of the following holds: i) the result of $a$ is only consumed by $b$: $\{(a, c) | (a, c) \in E_d, c \neq b\} = \emptyset$. If this constraint is met, only $b$ may have consumers outside the pair pattern or, ii) $b$ has no consumer: $\{(b, c) | (b, c) \in E_d\} = \emptyset$. If this constraint is met, only $a$ may have consumers outside the pair pattern.

• There is no path from $a$ to $b$ in $G$ other than $(a, b)$.

The 3-input constraint means the datapath only needs to supply one more source operand than for a normal RISC operation. The single external consumer constraint ensures that at most one result has to be written back, eliminating the need for an RF with multiple write ports. The last constraint ensures that when a pair in the DFG is transformed to a single node (as a special operation), the resulting DFG is still a DAG.
### Table 4.1: Kernel description.

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Description</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIR</td>
<td>5-tap finite impulse response filter</td>
<td>Image processing</td>
</tr>
<tr>
<td>Histogram</td>
<td>256-bin histograming</td>
<td>Image processing</td>
</tr>
<tr>
<td>YUV2RGB</td>
<td>YUV to RGB color space conversion</td>
<td>Image processing</td>
</tr>
<tr>
<td>IDCT</td>
<td>2D 8x8 Inverse cosine transformation</td>
<td>Codec</td>
</tr>
<tr>
<td>MatVec</td>
<td>Matrix vector multiplication</td>
<td>Linear algebra</td>
</tr>
<tr>
<td>CRC</td>
<td>Cyclic redundancy check code calculation</td>
<td>Network/Storage</td>
</tr>
<tr>
<td>DES</td>
<td>The Data Encryption Standard algorithm</td>
<td>Security</td>
</tr>
</tbody>
</table>

### Table 4.2: Percentage of DFG nodes covered by possible pair patterns.

<table>
<thead>
<tr>
<th></th>
<th>FIR</th>
<th>Histogram</th>
<th>YUV2RGB</th>
<th>IDCT</th>
<th>MatVec</th>
<th>CRC</th>
<th>DES</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>70.37%</td>
<td>62.68%</td>
<td>76.79%</td>
<td>67.43%</td>
<td>71.79%</td>
<td>93.51%</td>
<td>95.54%</td>
<td>76.00%</td>
</tr>
</tbody>
</table>

### 4.1.2 Pattern Frequency Analysis

The pair pattern described in Section 4.1.1 has a few constraints that can reduce the overhead and complexity of implementing special instructions in a RISC architecture. Another important design choice is how many patterns should be supported. In this subsection, the seven kernels in Table 4.1 are analyzed. These kernels are from different application domains, which give a good overview of the occurrence of pair patterns in various applications.

We analyze the data-flow graph (DFG) of each basic block in each kernel to find all the patterns that meet all the constraints. The percentage of DFG nodes covered by these pair patterns are shown in Table 4.2. On average, the specified pair patterns cover 76% of all the nodes in the DFGs. It means that even with the constraints, the pair patterns described in Section 4.1.1 are likely effective in various applications.

The number of patterns greatly affects the efficiency of the support of special instructions. Therefore the number of distinct patterns in different applications is important as the goal of this work is to support a flexible SFU in a generic embedded processor. Assuming the basic operation set has \( N \) operations, the total number of possible pair patterns is \( N^2 \), which is large. The number of distinct patterns needed by different applications is also large, as the applications vary in characteristics. For example, the IDCT kernel has a lot of fixed-point arithmetic oper-
Figure 4.2: Cumulative distribution of patterns in different kernels. For each kernel the patterns are sorted by number of possible matches in descending order. In each kernel, over 80% of the pattern matches occur in the 8 most frequent patterns.

operations, resulting in many possible patterns that contain multiplication and shifting operations. On the other hand, the CRC kernel has a lot of patterns with shifting and logical operations. Table 4.3 lists the number of distinct pair patterns found in each kernel, and the total number of distinct patterns in all kernels. The numbers suggest that although the number of possible patterns is large, each kernel usually only needs a few operation pairs to have a good coverage. Figure 4.2 shows the cumulative distribution of the most frequent pairs in each kernel. For all kernels, eight patterns are sufficient to cover over 80% of the possible pairs. This statistics suggests that, although in different kernels vary in pattern characteristics, the pair patterns have good locality in each kernel. In a specific function or kernel, the number of operation pairs is relatively low compared to the number of all possible pair patterns. To use pair patterns in applications from different domains, a processor needs to support a large number of pairs, but the size of the active pattern set can be very small if the pattern locality is properly exploited, which can reduce the design complexity.

4.2 Flexible Special Instructions in a RISC Processor with Compact ISA

The first step towards supporting the proposed operation pair patterns is to implement a hardware special function unit (SFU) that executes these patterns. We present the design of this SFU in Section 4.1.1. Another challenge is how to integrate the SFU into the datapath of a RISC processor with 24-bit compact ISA in an energy efficient fashion. We propose a low cost integration method in Section 4.2.2 to solve this problem.
CHAPTER 4. ENERGY EFFICIENT FLEXIBLE SPECIAL INSTRUCTIONS

4.2.1 Special Function Unit

The design of our special function unit (SFU) is shown in Figure 4.3. The SFU has two levels of sub-function-units (sub-FUs). An operand switch network is included in the SFU to allow more flexible operand encoding in a processor. The design of the SFU allows almost arbitrary operation pairs that meet the constraints specified in Section 4.1.1, except the ones containing branches and memory accesses.\(^1\) The sub-FUs at each level share the same input ports. Operand isolation is used to avoid unnecessary circuit activity in each sub-function-unit [2]. The input ports of each sub-FU are isolated by AND gates controlled by enable signals generated from the opcodes.

Due to the two levels of sub-FUs, the proposed SFU is likely to increase the delay of the critical path. When a processor with the proposed SFU needs to run at high frequency, there are two ways to mitigate this effect: i) add a pipeline register inside the SFU, either between the first and second layer, or at the output (or in the middle) of long latency units (e.g., the multiplier); ii) allow the long latency operations to run in multiple cycles, thereby allowing the SFU to run at a higher frequency. By using either method a processor with the proposed SFU is able to run at a frequency close to the one without the SFU.

To control the SFU, the following signals are needed: i) opcode for each level; ii) input data selection for each level; iii) sub-function-unit enable signal; iv) extra operand information, e.g., whether the immediate value should be sign-extended or zero-extended. When fully decoded, the control signal for the SFU shown in Figure 4.3 requires 18 bits.

\(^1\)Based on timing and area consideration, only one multiplier is included, which is put in level-1.
4.2.2 Integrating SFU into RISC Datapath

The proposed SFU is integrated into the datapath of a 4-stage RISC with 24-bit compact instruction set architecture (ISA). Table 3.1 lists the main features of the compact ISA. More details of the baseline RISC ISA are given in Appendix A. The major limiting factors of integrating the proposed SFU are:

- After adding basic operation, only less than 16 opcodes can be used for special instructions.
- For three-input instructions, at most three bits can be used to encode the extra operand, which is not enough for a register index.
- In each cycle, the RF can only supply two operands to the SFU.

The most straightforward solution is to increase the instruction width from 24 to 32 and the number of RF read ports from two to three. Figure 4.4 depicts the resulting datapath. Based on the estimation by CACTI [114], the energy consumption of the instruction memory increases by 10% to 30%, depending on the size and configuration. Similarly, the implementation results show that the RF energy consumption also increases by 12%. As the instruction memory and the RF are used by every instruction, the overhead of this straightforward solution is likely to make the architecture inefficient. Therefore we need to:

1. encode all possible operation pairs without increasing the width of the opcode;
2. supply the third source operand to the SFU with a two-read-port RF;
3. encode the third source operand with no more than 3 bits.

The analysis results in Section 4.1.2 show that many applications have good pattern locality. We propose a partially reconfigurable decoder that exploits the pattern locality to support a large number of patterns with only a few opcodes. Figure 4.5 illustrates the structure of this decoder. The main idea is to store the 18-bit SFU control signals in an 8-entry look-up-table called pattern table, which can be addressed with only eight opcodes. The content of the pattern table can be reconfigured according to the application requirement at runtime. Figure 4.6 show the

---

1The multiplier is absorbed into the SFU. So all basic operations executed by the multiplier in the baseline architecture are now handled by the SFU. Arithmetic and logical operations are still done in a separate ALU, which can isolate circuit activity more efficiently.
Instruction sequence for configuring the pattern table, which takes only 10 cycles in the worst case, thereby enabling fast dynamic reconfiguration.

To solve the operand problem, we use the explicit bypassing described in Chapter 3. By imposing the constraint that at least one of the source operands in a three-input instruction has to come from explicit bypassing, it is possible to supply three source operands to the SFU with a two-read-port RF. Because the number of bypassing sources is much smaller than the size of the RF (4 vs. 32 in this work), it takes only 2 bits to encode the extra operand from the bypass network.

Figure 4.7 depicts the datapath of the proposed architecture with partially reconfigurable decoder and explicit bypassing. In the proposed architecture, the use of special instructions has two constraints:

- For a three-input operation, at least one of the source operands has to come from the bypass network.
- At most eight special instruction patterns are active at a given moment. To use more patterns, reconfiguration of the pattern table is required.

Figure 4.8 shows an example of special instructions on different architectures. The proposed architecture achieves the same code size improvement as the one with 32-bit instruction and three-read-port RF.
Figure 4.7: Integrating SFU with explicit bypassing RISC.

Figure 4.8: Example of code with special instructions.
4.3 Code Generation for Reconfigurable SFU

The efficiency of the architecture described in Section 4.2 relies on a compiler. The key is to properly utilize the special instructions. To achieve high efficiency and productivity in various applications, we propose the design of a compiler for the proposed architecture. Figure 4.9 depicts the flow of the proposed compiler. The front-end is based on the open-source LLVM framework, which accepts the C-family high-level languages as input [87, 92]. The back-end is similar to the one introduced in Chapter 3.

The remainder of this section describes the back-end design in details: Section 4.3.1 introduces how the compiler detects and selects the aforementioned operation pair patterns to form special instructions. Section 4.3.2 discusses how instruction scheduling is performed for the proposed architecture. Lastly, Section 4.3.3 describes the remaining parts of the back-end, including register allocation and reconfiguration code placement.

4.3.1 Pattern Selection

To use the SFU, the compiler needs to choose basic operation pairs that can be used to form special instructions. In the pair pattern selection process, the compiler operates on the data-flow graphs (DFGs) of the basic blocks in the program. The first step of pair pattern selection is to find all the node pairs whose patterns are supported by the SFU in the DFG under the constraints described in Section 4.1.1. A set \( M \) containing all these node pairs is obtained by scanning through all nodes in the DFG. Each pair in \( M \) is called a match. The matches that are impossible to meet the operand bypass constraints are excluded from \( M \), e.g., the ones with three or more non-immediate live-in variables.

The next step is to choose a subset \( S \) of \( M \) that will be used to generate special instructions. Obviously each DFG node should only be used by one pattern in \( S \), as duplicating DFG nodes only results in extra energy consumption in the pair patterns. A match interference graph \( G_I(V,E) \) can be built:
• $V$ is a set of nodes, each representing a possible match in $M$.
• $E$ is a set of undirected edges between nodes in $V$. $(u, v) \in E$ means that $u$ and $v$ share a common DFG node. Hence $u$ and $v$ cannot be selected simultaneously.

An example of match interference graph is given in Figure 4.10: on the left is a DFG with four possible matches; on the right is the match interference graph of the four possible matches. $S$ should be an independent set of $G_I$, i.e., the nodes in $S$ are pair-wise non-adjacent in $G_I$. The objective here is to find as many pairs as possible, which is essentially to get the maximum independent set (MIS) of $G_I$, i.e., the independent set with maximum cardinality.

Though finding the MIS of a graph is NP-complete in general, the minimum degree heuristic performs very well for sparse and bounded degree graphs [54], which can be implemented in time linear in the number of edges and vertices. The algorithm used for pattern selection is depicted in Algorithm 4.1. In the DFG pair pattern selection, many nodes in the match interference graph have the same degree, which results in many ties in minimum degree selection. Since in the proposed architecture, only a limited number of operation patterns are supported without reconfiguration, the pattern frequency is used to break the ties. The frequency $F_p : V \rightarrow \mathbb{Z}^+$ defines the number of occurrences of each pattern, with each occurrence weighted by $10^{\text{loop depth}}$ to estimate the dynamic frequency. The algorithm yields $\{1, 3\}$ for the example in Figure 4.10, which is the MIS of the interference graph.

### 4.3.2 Energy-Aware Scheduling

A basic block list scheduler is used for the proposed architecture. In the proposed architecture, the total number of physical registers is reduced as part of the RF address space is used by bypass sources. So although explicit bypass eliminates the need for many temporary registers, it is still very important for the compiler to make sure that register pressure stays low. When the list scheduler greedily chooses the node with a maximum number of bypasses, the register pressure may go up. Figure 4.11 shows an example of how a greedy bypass scheduler may increase the register pressure. Here the live ranges of $r3$ and $r4$ are longer if the scheduler decides to select the multiply instruction that can use bypass value. When register pressure is
Algorithm 4.1: Pattern Selection

**Input:** Match interference graph \( M(V,E) \) and pattern frequency \( F_p \)

**Output:** Set of pattern matches \( S \) in which nodes do not interfere with each other

1. \( S \leftarrow \emptyset \)
2. while \( M \neq \emptyset \) do
3. \( D \leftarrow \{d | d \in V, \exists u \in V: \text{degree}(u) < \text{degree}(d)\} \) // Minimum degree nodes
4. if \( |D| = 1 \) then
5. \( n \leftarrow D[0] \)
6. else // Tie-breaking with pattern frequency
7. \( Q \leftarrow \{q | q \in D, \exists u \in D: F_p(u) > F_p(q)\} \)
8. // Pick the first one if \( Q \) has more than one node
9. \( n \leftarrow Q[0] \)
10. end
11. \( S \leftarrow S \cup \{n\} \)
12. Remove \( n \) from \( M \), along with all its edges and neighbors
13. end

high, the add instruction that uses R3 and R4 would be selected.

In this work we use a scheduling algorithm which is similar to the integrated prepass scheduling (IPS) [46]. The details of the algorithm are given in Algorithm 4.2. Depending on the register pressure of the current partial schedule and the number of available registers in the block, the scheduler switches between two policies:

- When the number of live variables is below the threshold value (number of available registers), choose the node that minimizes energy.
- When the number of live variables is above the threshold value, choose the node that minimizes register pressure.

![Figure 4.11: The effect of bypass scheduler on register pressure.](image)
The threshold value is the number of available registers for the basic block. The procedure `find_node_with_most_energy_gain` in Algorithm 4.2 tries to select the node with the most energy saving from the ready operation set. In this work, three possible energy saving scenarios are considered in the node selection:

- Making sure that a special operation can meet the constraints saves one instruction. This leads to energy savings in both the core and the instruction memory.
- Operand bypassing (including dead writeback elimination) can reduce the energy consumption of the register file.
- Keeping the same opcode as the previous instruction results in less circuit switching activities in the instruction decoder and FU.

Clearly choosing a special operation is the most beneficial scenario. The energy saving provided by operand bypassing alone is an order of magnitude less than special operation, because it only reduces the energy for accessing register file, which consumes much less energy than instruction memory accesses (from a few times to over an order of magnitude, depending on the exact configuration [10, 142]). And keeping the opcode unchanged in most cases results in the least energy saving among the three. So the scheduler determines the priority of the ready operations by the following rules:

- the scheduler first tries to select a special operation node that constraints are guaranteed to be met;
- if it fails to select a unique candidate, or no such special operation node exists, the scheduler chooses the node that benefits most from bypassing;
- if there are still multiple candidates, the scheduler chooses the node with the same opcode as the previous operation;
- if all tie breaking does not work, the scheduler chooses the first node it finds.

### 4.3.3 Finalizing Generated Code

After the scheduling, a scan through all instructions is performed to check for invalid special instructions, i.e., the instructions that do not meet the constraints given in Section 4.2.2. If a special instruction is found to violate any of these constraints, it is considered to be invalid and is decomposed into a sequence of normal instructions. Due to the nature of explicit bypassing, this transformation does not increase register usage.

The register allocation is done with a graph-coloring algorithm. The register allocation is almost the same as the one used for a normal RISC processor. One of the main differences is that the compiler may choose to keep small constant values (ones that can fit in the instruction immediate field) in registers for a special instruction when it requires two immediates, because only one immediate value can be encoded in the immediate field.

Afterwards the compiler collects pattern information and decides where to insert the reconfiguration codes. In this work, there are two possible scenarios:
Algorithm 4.2: Basic Block Scheduling

Input: DFG $G(V, E_d, E_f)$, live-in set $L_I$, live-out set $L_O$ and register threshold $t_r$

Output: The schedule of the DFG $T : V \mapsto \mathbb{N}$

```
1 $R \leftarrow \emptyset$ // Ready set
2 $L \leftarrow L_I$ // Live variable set
3 $S \leftarrow \emptyset$ // Set of scheduled operations
4 $c \leftarrow 0$ // Cycle counter
5 while $|S| \neq |V|$ do
6     if $|L| < t_r$ then
7         $o \leftarrow \text{find_node_with_most_energy_gain}(R, T)$
8     else
9         $o \leftarrow \text{find_node_reduces_most_register_pressure}(R, T, L, G)$
10    end
11    for $s \in \{u | u \in V, (o, u) \in E_d \cup E_f\}$ do
12        if $s$ is enabled by $o$ then
13            $R \leftarrow R \cup \{s\}$
14        end
15    end
16    for $p \in \{u | u \in V, (u, o) \in E_d\}$ do
17        // Check if live-range of $p$ ends
18        if $o$ is last use of $p$ and $p \notin L_O$ then
19            $L \leftarrow L \setminus \{p\}$
20        end
21    end
22    // Creates a new live variable
23    if $o$ has value output then
24        $L \leftarrow L \cup \{o\}$
25    end
26    $S \leftarrow S \cup \{o\}$
27    $T[o] \leftarrow c$
28    $R \leftarrow R \setminus \{o\}$
29    $c \leftarrow c + 1$
30 end
```
• If the number of patterns used in a function is less than or equal to the pattern table size, all patterns are loaded at the entry block of the function.
• If the number of patterns used in a function exceeds the pattern table capacity, the compiler tries to insert reconfiguration code before each intensive loop such that the patterns used in the loop can be put in the table. The loop information can be obtained through static estimation or profiling.

When both ways fail to accommodate all used patterns, the compiler selects the most frequently used patterns. And a special instruction whose pattern is not in the pattern table is decomposed to two normal instructions.

As shown in Figure 4.9, whenever a code transformation changes the schedule, the bypass status of each instruction needs to be updated, so the validation of the special instructions needs to be repeated. This process terminates: in the worst case, the loop stops when all special instructions are decomposed to normal instructions. In practice, one to two iterations are sufficient in most cases.

In the proposed design, the scope of the pattern table is within a function. This might introduce overhead across function calls. But the frequently called simple functions usually get in-lined when compiler optimization is enabled. And since the cost of configuring the pattern table is only 10 cycles, we expect that the reconfiguration overhead is negligible in most cases. When there is a function call, the pattern table becomes part of the context, and needs to be treated in a similar way as the general purpose registers. In the proposed toolflow, the caller is responsible for maintaining the table values. The main reason is that in the proposed flow, the pattern table entries are constant values generated at compile time. So the caller does not need to save the values. If the table is callee-saved context, the callee needs to actually store the table values to the stack, which results in larger overhead. The results in Section 4.4 show that the reconfiguration overhead is indeed quite small.

4.4 Experimental Results

A fully functional processor and a compiler based on the proposed design are implemented. To compare it with alternative approaches, we run experiments on four different processors. Table 4.4 lists the architectures used in the experiments. The proposed architecture, i.e., the one with partially reconfigurable decoder, explicit bypass network, and constrained special instruction patterns (see Section 4.2.2), is called SFU-I24. And the architecture that integrates SFU without the constraints introduced in SFU-I24 is called SFU-I32. The SFU-I24-C2 is an architecture that is almost identical to SFU-I24, except that a two-cycle SFU and interlock logic are used to achieve higher frequency (see Section 4.4.1). The datapaths of the baseline, SFU-I32, SFU-I24/SFU-I24-C2 are shown in Figure 2.4, Figure 4.4 and Figure 4.7, respectively.

All four cores are implemented in Verilog HDL and synthesized with the TSMC 90nm low power library at 1.2V and typical case. Clock gating is used to reduce the dynamic power consumption. The core energy consumption is estimated with the back-end information and
Table 4.4: Configuration of different architectures

<table>
<thead>
<tr>
<th></th>
<th>Baseline (Base)</th>
<th>Unconstrained SFU (SFU-I32)</th>
<th>Proposed (SFU-I24)</th>
<th>Proposed w/ 2-cycle SFU (SFU-I24-C2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Width</td>
<td>24 bits</td>
<td>32 bits</td>
<td>24 bits</td>
<td>24 bits</td>
</tr>
<tr>
<td>Instruction Memory</td>
<td>12kB 24-bit</td>
<td>16kB 32-bit</td>
<td>12kB 24-bit</td>
<td>12kB 24-bit</td>
</tr>
<tr>
<td>Data Memory</td>
<td></td>
<td></td>
<td></td>
<td>4k instruction words</td>
</tr>
<tr>
<td>Register File</td>
<td>32b×232</td>
<td>32b×32</td>
<td>32b×28</td>
<td>32b×28</td>
</tr>
<tr>
<td>SFU Patterns</td>
<td>0</td>
<td>128</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Two-cycle Special Operations</td>
<td></td>
<td>None</td>
<td>Special operations with multiplication</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.5: Memory energy consumption

<table>
<thead>
<tr>
<th>Memory</th>
<th>16kB 32-bit</th>
<th>12kB 24-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Energy per access (pJ)</td>
<td>15.38</td>
<td>11.62</td>
</tr>
</tbody>
</table>

the real toggle rate generated by post-synthesis simulation with clock frequency of 100MHz. The area and energy consumption of the memory are estimated with CACTI [114], using 90nm low operating power technology. Table 4.5 shows the energy model of the memory used in the experiments.

4.4.1 Area and Frequency

The implementation results of the four architectures are shown in Table 4.6. The increase in the core area of the architecture with SFU is expected, as the SFU and its decoder are much more complex. The core area of SFU-I32 is slightly larger than SFU-I24 as it needs to support much more active patterns. SFU-I24-C2 uses slightly more area than SFU-I24 because of the interlock logic for multi-cycle operations. SFU-I32 has much larger memory area, which is caused by the instruction memory since SFU-I32 uses 32-bit instructions. Unlike SFU-I32, the proposed SFU-I24 and SFU-I24-C2 realize the special instruction support with a relatively small overhead. In particular, they do not increase the memory area, which is a dominant part in many modern processors.

The reduced maximum frequency of SFU-I32 and SFU-I24 is mainly caused by the single-cycle SFU. In SFU-I24-C2, this is mitigated by making some special operations run in two cycles. Compared to the baseline, there is still a 14.4% loss in frequency, which is primarily caused by the operand switch network in the SFU (see Figure 4.3).
Table 4.6: Implementation result comparison

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Base</th>
<th>SFU-I32</th>
<th>SFU-I24</th>
<th>SFU-I24-C2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normalized Core Area</td>
<td>1.000</td>
<td>1.309</td>
<td>1.268</td>
<td>1.278</td>
</tr>
<tr>
<td>Normalized Memory Area</td>
<td>1.000</td>
<td>1.154</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>Maximum Frequency (MHz)</td>
<td>450</td>
<td>325</td>
<td>325</td>
<td>385</td>
</tr>
</tbody>
</table>

Table 4.7: Results of the baseline architecture

<table>
<thead>
<tr>
<th></th>
<th>Simulated Cycles</th>
<th>Average Core Energy per Cycle (pJ)</th>
<th>Average Memory Energy per Cycle (pJ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Histogram</td>
<td>21547</td>
<td>11.05</td>
<td>16.10</td>
</tr>
<tr>
<td>FIR</td>
<td>40973</td>
<td>18.41</td>
<td>16.24</td>
</tr>
<tr>
<td>IDCT</td>
<td>2303</td>
<td>17.93</td>
<td>14.56</td>
</tr>
<tr>
<td>YUV2RGB</td>
<td>43032</td>
<td>17.88</td>
<td>13.82</td>
</tr>
<tr>
<td>MatVec</td>
<td>3729</td>
<td>13.27</td>
<td>14.00</td>
</tr>
<tr>
<td>CRC</td>
<td>162017</td>
<td>12.73</td>
<td>11.82</td>
</tr>
<tr>
<td>DES</td>
<td>857130</td>
<td>14.89</td>
<td>14.64</td>
</tr>
</tbody>
</table>

4.4.2 Energy Consumption

The benchmark applications used in the experiments are listed in Table 4.1. The compiler described in Section 4.3 compiles the kernels written in C into machine codes for different targets with maximum optimization level (-O3). For SFU-I32, the code generation process is almost the same as SFU-I24, except that all the constraints on operand bypassing and opcode space are removed, and no reconfiguration code is generated. Table 4.7 shows the absolute results of the baseline processor (Base). Note that the memory energy in the table includes accesses to both instruction memory and data memory. In the remainder of this subsection, we normalize all results to the baseline.

![Figure 4.12: Dynamic cycle count including overhead.](image)
Figure 4.12 shows the normalized cycle count of the four different processors, including the reconfiguration overhead in SFU-I24 and SFU-I24-C2. SFU-I24 achieves a reduction of 25.9%, which is only 2.1% worse than SFU-I32, which supports all operation pairs without reconfiguration. In some kernels that need to use special operations with multiplication, SFU-I24-C2 uses more cycles. But on average it still reduces 24.8% in cycle count compared to Base. When the instruction width is factored in, as shown in Figure 4.13, SFU-I32 only reduces 3.5% memory energy on average. Though it fetches much fewer instructions, the increased instruction width makes it difficult to reduce the energy dramatically. In 3 out of 7 benchmarks the energy consumption actually goes up. In contrast, the proposed SFU-I24 is able to directly convert the reduction in instruction count into memory energy saving. On average SFU-I24 saves 21.7% memory energy. Similarly, SFU-I24-C2 achieves an average saving of 21.3%.

Figure 4.15 shows the normalized core energy consumption. Comparing to Base, the pro-
posed SFU-I24 reaches a maximal core energy reduction of 21.5% in the FIR case, and of 11.2% on average. The main contributions of energy reduction are from: 1) reduced RF access energy; 2) reduced datapath and control path overhead due to special instructions. On the other hand, SFU-I32 increases the average core energy by 1.7%, and it performs badly in two cases: FIR and IDCT, in which the core energy increases by over 8%. The explicit bypass network is an important contributing factor in this huge difference. As shown in Figure 4.14, the number of accesses to the RF in SFU-I24 is significantly reduced. In addition, the RF in SFU-I24 has fewer ports than the one in SFU-I32. As a result, the core of SFU-I24 consumes much less energy compared to SFU-I32. In SFU-I32, a degradation of over 5% is observed in both FIR and IDCT. In the case of SFU-I24-C2, the core energy consumption increases when a lot of special instructions with multiplication are used. The main reason is that the compiler is forced to use a less energy efficient schedule in order to fill the delay caused by multi-cycle operations. When no special instruction with multiplication is used, the result is similar to SFU-I24. On average, the core energy is reduced by 5.8% compared to the baseline.

Figure 4.16 shows the normalized total energy consumption. The proposed SFU-I24 re-
duces both the memory and core energy, and it achieves an average saving of 15.8%. It reaches a maximal of 33.1% energy saving in CRC. In SFU-I24-C2, the total energy is reduced by an average of 13.1%, and the maximal saving occurs in CRC, which is 32.2%. While in the case of SFU-I32, the total energy saving is only 1.1%. The breakdown of the energy in different kernels is shown in Figure 4.17, which clearly shows that SFU-I24 and SFU-I24-C2 outperform SFU-I32 in both core and memory energy in most cases.

These results show that although the use of an SFU is able to significantly reduce the dynamic cycle count, directly putting the SFU into a generic processor without any constraint does not result in an energy efficient architecture.

4.4.3 Performance

The normalized execution time of the different cores used in the experiments is shown in Figure 4.18. The result is calculated based on the dynamic cycle count and the maximal frequency of each core. Due to the loss in frequency, both SFU-I32 and SFU-I24 suffer minor performance degradation, though they are able to reduce the cycle count by about 25%. However, in SFU-I24-C2, an average speed-up of $1.14\times$ is observed. The SFU-I24-C2 is able to achieve a good balance between performance and energy consumption, with relatively small overhead compared to SFU-I24.

In summary, the proposed architecture with a partially reconfigurable decoder and an explicit bypass network is able to reach a balance between the energy efficiency and the flexibility of the SFU, and it results in an architecture with high energy efficiency and good performance.
4.5 Related Work

The use of complex operation patterns, called instruction set extension (ISE), is common in instruction set synthesis for application specific and reconfigurable processor design [76, 24, 90]. A comprehensive survey that covers most recent works in ISE can be found in [41]. The idea of using a dynamically reconfigurable decoder can be found in some early high-performance architectures, in the form of programmable microcode [125]. The R.E.A.L DSP supports Application Specific Instruction (ASI), which use a 256-entry table to extend 16-bit instructions to 96-bit VLIW instructions [86]. Reconfigurable architectures like Montium [60] and MOLEN [152] exploit reconfigurable decoders to support flexible ISAs, but they are not tightly integrated into the datapath of general purpose processors. The toocllchain of Conservation Cores uses automatically synthesized accelerators called c-cores to improve energy efficiency in many core architectures [153]. The ConCISe toolchain proposed by Kastrup et al. introduces accelerator called reconfigurable function unit (RFU) based on programmable logic device (CPLD/FPGA). The RFU is integrated into a MIPS datapath as a new FU [78]. Similar ideas could be found in the ADRES reconfigurable processor [96], the Stretch software configurable processor [118, 44], the rotating instruction set processing platform (RISPP) [14] and the rASIP [75] integrate coarse-grained reconfigurable components into the datapath of baseline processors. Venkatesh et al. proposed QsCores, a framework that automatically synthesizes accelerator for a wide range of applications from source code [154]. In the aforementioned works, relatively expensive reconfiguration is usually required and the cost to support flexible ISE is high compared to this work. The FITS framework [22] explores instruction set tuning to optimize instruction encoding and operand bandwidth in embedded processors. Compared to this work, FITS put more focus on how to tailor ASIPs for specific applications.

There are also studies trying to integrate ISE in general purpose architectures, most of which focus on improving performance. Clark et al. proposed integration of a configurable compute accelerator (CCA) into a general-purpose processor [26, 25]. Compared to the SFU in this chapter, CCA is relatively complex and it requires up to 4 inputs and 2 outputs. The control part of CCA is designed to be transparent so that the code can be executed with or without CCA.
Woh et al. proposed AnySP, a wide SIMD signal processor targeting wireless and multimedia applications [161]. In AnySP the idea of operation pairs is similar to the SFU design in this work, but the operand bandwidth problem is partially solved by introducing an extra small RF. In PEPSC, an architecture designed for efficient scientific computing, Dasika et al. proposed an FPU that is capable of executing up to five back-to-back operations [31]. The Intel Pentium M processor performs micro-op fusion, which combines some dependent pairs of micro-ops dynamically [42]. The micro-op fusion in Pentium M achieves a typical performance increase 5% for integer code and 9% for floating-point code. Bracy et al. proposed dataflow mini-graph, which exploits sub-graphs of dataflow graphs that contains two inputs and one output on superscalar processors with Mini-Graph Table (MGT) [17]. A binary rewriting tool is used to discover mini-graphs in executables and fill the MGT. The concept of the MGT is similar to the patten table in this work, but it is more complex as it needs to facilitate dynamic scheduling. Sassone and Wills presented Dynamic Strands, a method to dynamically detect and chain dependent operations without intermediate fan-out in superscalar processors [135]. Di Biagio et al. proposed a method to support operation pair patterns by utilizing existing adder in the decoding stage of a processor [16]. Compared to the architecture in [16], the pair pattern supported in this work is more flexible, and it does not require additional RF ports. And unlike in [16], the compiler is aware of the pair pattern, and is able to optimize for it.

The data bandwidth from the register file to the FUs is an important constraint in ISE design [5]. Leupers et al. introduced special register file called internal registers (IR) for the SFUs [90]. The IR is an effective way of implementing application specific special instruction, but it lacks flexibility and complicates the code generation as the registers and FUs are no longer orthogonal. Karuri et al. proposed RF clustering in single issue processor to mitigate the register file port pressure in ISE in ASIP design [77]. While reducing port pressure, the RF clustering, which is similar to what is used in clustered VLIW architectures, also makes the code generation much more complex. Pozzi and Ienne exploited the fact that pipelined SFUs do not need all operands in the same cycle to distribute register file accesses across multiple cycles [110]. This cannot be applied to the SFUs that are similar to the one used in this work. Utilizing the bypass network has been proven to be an efficient way to increase operand bandwidth and reduce register file energy in different types of architectures [10, 11, 107, 139, 140, 142]. Jayaseelan et al. proposed explicit forwarding to reduce register file port pressure and operand encoding cost for application specific ISE in a RISC-like datapath, which resembles the idea of explicit bypass in this work [71]. However the power model used in [71] only considers the consumption of the register file. The overall energy efficiency of the proposed architecture is not clear. Cong et al. proposed shadow registers to solve the operand bandwidth issue for supporting special instructions in a configurable processor [28]. The shadow registers are similar to explicit pipeline registers, but have more flexibility. To avoid dramatic increase of control bits, the shadow registers are hash-mapped, which may be less efficient in terms of energy. In this chapter, we explored the trade-offs in utilizing bypass network for energy efficient ISE in a generic processor with compact ISA, along with detailed and realistic results. The proposed solution achieved high energy efficiency while maintaining the generality.
Code generation is key in supporting ISE for both application-specific and general purpose processors. Selection and scheduling for special instructions is one of the core aspects in code generation for ASIP and reconfigurable architectures. There are quite a few works on identifying and synthesizing special instructions for ASIPs [5, 6, 167]. In this work, the special instruction generation problem is limited to the generation of operation pairs. So we focus on selecting the patterns that can meet the constraints and fit in the pattern table. Scheduling algorithms that are aware of the bypassing is another key element for generating code for the proposed architecture. Park et al. presented a greedy algorithm for increasing the bypassing in a RISC processor [107]. For architectures that fully expose bypass network to software, like TTA and STA, bypass aware scheduling can achieve significant reduction in RF traffic, thereby improving performance and energy efficiency [50, 142]. But the fine-grained control over the datapath also introduces overhead in code density, resulting in increased fetching and decoding energy. In [71], integer linear programming (ILP) is used to perform bypass aware scheduling in a processor with application specific ISE. The proposed algorithm inserts register copying instructions in order to meet the constraints of the special instructions. In this work, we propose a modified list scheduling algorithm, in which the priority calculation takes into account the energy impact of special instructions as well as explicit bypassing. The results show that the proposed algorithm is effective.

In this chapter, we introduced a novel architecture that uses special instructions. In contrast to above works, this work aims at improving the energy efficiency of a generic processor with a compact ISA. Two major issues: i) opcode and operand encoding and ii) operand bandwidth to SFU are solved by using a partially reconfigurable decoder and explicit bypass network.

4.6 Summary

Special instruction support is a well-know method to improve processor performance and efficiency. However, efficiently integrating a Special Function Unit (SFU) that executes flexible complex operations into a generic processor is challenging, as supporting special instructions incurs large overhead, especially when the ISA is compact. This chapter introduced an architecture for integrating an SFU that supports flexible operation pair patterns in a generic processor with a compact ISA. A partially reconfigurable decoder and a software-controlled explicit bypass network are used to: i) encode special operation opcodes and operands within the limited coding space; ii) supply sufficient data to the SFU without increasing the number of register file ports. We presented a compiler back-end design for the proposed architecture. The compiler is able to utilize the SFU and the explicit bypass network to generate energy efficient code. Results including benchmarks from different domains are promising: average dynamic cycle count is reduced by over 25%. The total processor energy consumption is reduced by 15.8%. When high performance is required, the proposed architecture is able to achieve a speed-up of $1.14 \times$ with 12.6% energy reduction compared to the baseline, by introducing multi-cycle SFU operations. All in all, the proposed architecture and compiler reach a good balance between flexibility and efficiency, resulting in an energy efficient processor design.
Energy efficiency is becoming a critical design factor in high performance embedded processors, especially if they are running on limited power sources like batteries. In last two chapters we discuss means to reduce overhead in RISC-like processors via explicit bypassing and flexible special instructions. However, the control overhead, including instruction fetching and decoding, is still a large contributing factor of inefficiency. In this chapter, we explore reducing the control overhead via exploiting the abundant data-level parallelism (DLP) contained in many emerging embedded streaming applications. The Single Instruction Multiple Data (SIMD) architecture is able to perform the same operation on multiple data items simultaneously, thereby providing high computational throughput with low control overhead, and enabling the processor to get closer to the intrinsic energy consumption. Therefore using processors based on wide SIMD architecture in such applications is a promising solution [57, 161].

In wide SIMD processors, register files (RFs) are again among the most energy consuming components, in particular because the relative size of the control-path is reduced considerably compared to other types of processors [161, 156]. In this chapter, we propose to use a wide SIMD architecture with an explicit datapath to reduce the RF energy consumption. By using an explicit datapath that allows the software to directly control the data bypassing, accesses to the RFs can be dramatically reduced. Hence the processor energy consumption is brought closer to the intrinsic energy consumption, i.e., the part of energy that is consumed by useful computation.

An efficient compiler is a key to efficiently utilize the proposed architecture. Code generation for SIMD processors has always been one of the most difficult problems in compiler design. The open computing language (OpenCL) is a standardized language for programming heterogeneous parallel platforms. Initially it is designed for General Purpose Graphics Processing Units (GPGPUs) [111, 160], some of which are also based on wide SIMD architectures. It is also suitable for programming low-energy SIMD processors. However, there are challenges to use OpenCL for the proposed architecture in this chapter:

- Compared to the flexible crossbar and network in many commercial GPGPUs [104, 160], the interconnect between PEs and the memory system in the proposed architecture only
allows a limited form of communication between PEs, which makes efficient mapping of OpenCL kernels challenging.

- The bypassing in the PE datapath is directly controlled by the software, which means the compiler has to handle that in order to generate correct and efficient code.

In this chapter, we propose the design of a compiler that is able to analyze the OpenCL kernels with statically analyzable memory accesses and to map them onto the proposed architecture. After mapping OpenCL kernels, the compiler performs bypass-aware scheduling to efficiently utilize the explicit datapath in the proposed architecture.

The proposed architecture is implemented in synthesizable RTL. Five kernels developed in OpenCL are tested on processors with different configurations. Energy consumption of the processors is accurately estimated using a commercial ASIC toolflow. The results of the proposed architecture are compared against a RISC processor and an SIMD architecture without explicit bypassing. The results show that the proposed architecture is scalable, and is able to achieve high energy efficiency. For a 128-PE configuration, the average speed-up is $85 \times$ compared to the RISC baseline, which is the same as the non-explicit bypassing SIMD processor. And on average, the proposed architecture with 128 PEs reduces the total energy consumption by 49.5%, which is 33% less than the SIMD processor with automatic bypassing.

The remainder of this chapter proceeds as follows: The proposed architecture is described in Section 5.1. Section 5.2 introduces the compiler design and the mapping of OpenCL kernels in the proposed architecture. Experimental results that show the effectiveness of the proposed design are given in Section 5.3. Related work is discussed in Section 5.4. Lastly, Section 5.5 concludes our findings in this chapter.

### 5.1 Low-Energy Wide SIMD Architecture

The wide SIMD processor architecture used in this work is based on the one in [141, 156]. The proposed processor architecture consists of two parts, a control processor (CP) and a wide one dimensional (1-D) array of processing elements (PEs), which run in lock-step. It results in a VLIW processor with one scalar issue slot for the CP and one vector issue slot for the PE array. Figure 5.1 depicts the proposed architecture.

#### 5.1.1 Explicit Datapath

In the proposed architecture, the PEs use a datapath with explicit bypassing, which is similar to the one described in Chapter 3. The datapaths of the 4-stage and 5-stage PE are shown in Figure 5.2. Different number of pipeline stages provides trade-offs between area, performance and power. The 4-stage datapath is the same as the one described in Chapter 3. Compared to conventional architectures, instructions in the proposed architecture have more control over input operands and destinations:

- An instruction directly specifies whether an input operand is from RF or one of the by-
passing sources.

- Each function unit (FU) in the datapath has separate input registers that ensures the result of the FU remains stable until the next operation is issued to the FU, resulting in more bypassing opportunities.

- Each instruction can control whether the result needs write-back. Three options are available:
  - No write-back: the result is only available at the FU output;
  - To WB stage: the result is written to the pipeline register in the write-back stage, but not to the RF;
  - To RF: the result is stored into the RF.

To use explicit bypass without changing the instruction format, part of the RF index space is used for the bypassing sources. As a result, the number of registers in the RF is reduced from 32 to 28 (4-stage) or 27 (5-stage). The impact of a smaller RF is mitigated by the fact that there is no need to allocate registers for short-live variables in many cases.

### 5.1.2 Circular Neighborhood Communication Network

Unlike SIMD architectures with small vectors, a fully connected shuffling network is not scalable in a wide SIMD processor. In the proposed architecture, a one dimensional (1-D) neighborhood network is used. Figure 5.3 shows the architecture of the network. Each PE can only communicate with its left and right neighbors by using the value from its neighbors’ RF or bypass network as the source operand in the decode stage. The boundary PE (the first or the last PE) selects the source operand based on the boundary mode, which is configured by setting the value of a memory-mapped special register. There are four available boundary modes:

- **Zero**: the boundary PE gets zero.
Figure 5.2: Datapath with explicit bypassing. The 4 stage datapath is the same as the one described in Chapter 3. The ALU in the 5 stage datapath has two output stages (named \textit{ALU1} and \textit{ALU2}), and the result of an operation flows through both stages regardless of the bypass option.

Figure 5.3: Circular neighborhood communication network.

- \textit{Self}: the boundary PE gets the value from its own RF or bypass network.
- \textit{Wrap}: the first PE gets the value from the last PE, or vice versa. Note that the network becomes circular in this mode. But the connection between the first PE and the last PE does not introduce long wires, since in physical layout the PEs can be placed in a circular manner.
- \textit{Scalar}: the boundary PE gets the value from the CP’s RF or bypass network. As illustrated in Figure 5.3, CP becomes an extra node in the communication network in this mode, allowing data exchange between the scalar datapath and the vector datapath.

In addition, the CP is able to broadcast data to all PEs, allowing the CP to perform calculations that are common to all PEs, which could be more energy efficient.
5.1.3 Instruction Set Architecture and Instruction Format

The ISA of both the CP and the PE is based on a 24-bit RISC ISA similar to the one described in Chapter 3. Some modifications are made to improve the support of OpenCL parallel applications:

- Only CP may execute control instructions such as jump and conditional branch.
- The first source operand (Op A) of each PE instruction may come from one of: i) local RF/bypass, ii) left or right neighbor, iii) CP broadcasting. Figure 5.3 illustrates the source selection for the first operand.
- A two-entry predicate register file is introduced. Each instruction may have zero, one or two predicates. If an instruction is predicated, it is only executed if all its specified predicate registers are set. The value of a value register can be set by compare instructions. The predicate register file simplifies the control flow mapping and is helpful in mapping OpenCL kernels.

As illustrated in Figure 5.4, the instruction format is changed for the proposed architecture to facilitate the changes. Four bits are added: two for encoding the communication, and two for addressing the predicate register file. As a result, the instruction width is 28 bits. The size of a 2-issue instruction packet containing one CP instruction and one PE instruction is 56 bits. An example packet is shown in Figure 5.4. The CP instruction specifies that it is reading from the first PE (the prefix h. stands for head) for the first source operand. The communication between CP and the first PE is performed in the same way as shown in Figure 5.3, i.e., CP gets the value of the first source operand of the first PE, and vice versa. In the example in Figure 5.4, the CP gets the ALU output value of the first PE as its first operand. Similarly, each PE gets the ALU output value of its right neighbor as the first operand. In addition, only the PEs in which the value of P1 is 1 will execute the multiply instruction.

5.1.4 Accelerator Architecture

Figure 5.5 depicts a generic architectural template for using the proposed architecture as an accelerator. The slave interface allows the host to have direct access to all the instruction and data memories of the accelerator. For more efficient data transfer, a direct-memory-access (DMA) controller is used to move data between external memory and the memories of the
5.2 Code Generation for Wide-SIMD

Compiler is key to efficiently utilize the proposed wide SIMD architecture. The standardized OpenCL language is ideal for such architectures. In this work we propose to use OpenCL as the programming language for the proposed architecture. A complete OpenCL program consists of two parts:

1. One or more OpenCL kernels to perform computation on compute devices.
2. A standard C program that runs on a host processor. The host program calls OpenCL APIs to control the behavior of the kernels [116].

On the platform depicted in Figure 5.5, the host program runs on the host processor, and the kernels run on the accelerator based on the proposed wide SIMD architecture.

Figure 5.6 shows the process to compile an OpenCL program that runs on the proposed platform. The host code is compiled by the native compiler of the host processor. The device code is compiled by a retargetable compiler designed for the proposed architecture. Figure 5.7 shows the flow of the device compiler with OpenCL support. The front-end of the compiler is based on the open-source LLVM compiler framework [87, 92]. The front-end produces a low-level intermediate representation (IR). The back-end processes the IR and the target specification, and produces the machine code.

The remainder of this section gives more details about the compiler back-end. Section 5.2.1 introduces how to map an OpenCL program onto the proposed architecture. The analysis and optimizations for interleaved work-items and data mapping are described in Section 5.2.2. Finally, Section 5.2.3 presents the reset of the code generation for the proposed architecture, including legalizing communication, instruction scheduling and register allocation.

5.2.1 Basic OpenCL Kernel Mapping

Explicit loops are created on the CP to execute all work-items in an OpenCL kernel. Different work-groups are executed sequentially. The work-items in a work-group are mapped to the PE array. For work-groups with a size that is not aligned to the number of PEs, predication is used.
CHAPTER 5. EXPLOITING DATA-LEVEL PARALLELISM

Figure 5.6: Compilation process for OpenCL programs.

Figure 5.7: Compiler back-end flow.
Figure 5.8: Converted OpenCL kernel, loops are inserted whenever necessary.

Figure 5.9: Vector memory address space and linear mapping for an N-PE system.

to guard the execution. Figure 5.8 illustrates how the control flow of a kernel is converted. For work-group with synchronization barriers, this work uses a similar approach as [67]. The work-group loop is split at each barrier, which ensures that all work-items finish the work prior to the barrier before continuing.

Memory mapping is an important step in mapping OpenCL kernels. Figure 5.9 shows the address space of the PE data memory in the proposed architecture. Table 5.1 shows the mapping of different address spaces in OpenCL to the memory hierarchy of the proposed architecture. The private memory space is only accessible by each work-item. Since in the proposed architecture the vector data memory is divided into banks that each can only be accessed by one PE (see Figure 5.1), it is natural to map data in the OpenCL private memory to the memory bank of each PE. The data in OpenCL global and local memory space is also mapped to the vector data memory, and when a PE needs to access a data item stored outside its memory bank, it needs to do it via the communication network. By default, the compiler maps global and local data arrays linearly onto the vector data memory, i.e., in row-major order. The method to determine the physical location of an address in a linearly-mapped array is shown in Figure 5.9. If the launch parameters and memory accesses in the kernel can be statically analyzed, the compiler calculates the exact location and the inter-PE communication required for each access. A series of data shifting via the neighborhood network is inserted if the data needed by a PE is not in the memory bank of that PE. The distance of the communication can be optimized by changing the memory mapping, which is described in Section 5.2.2. The data in the constant memory can
be mapped to the CP memory if all PEs access the same data in each loop iteration. The CP broadcasts the constant values to all PEs at runtime. Otherwise it is mapped as constant data in the memory banks of each PE.

To generate code for a complete OpenCL program for the system shown in Figure 5.5, the proposed toolflow requires the complete sources of both the host and the device to be available at compile time. The host processor controls the kernel launching. For each kernel, the host processor sends the launch parameters to the accelerator. The input data and instructions are copied to the local memory of the accelerator by the DMA controller. After the kernel execution, the output data can be copied to external memory by the DMA controller.

**Code Generation for Generic Communication**

If the compiler fails to analyze the communication parameters statically, generic communication is required. The loop for a store with generic inter-PE communication, of which the communication distance is unknown and variable, is shown in Algorithm 5.1. Figure 5.9 shows how the PE index $C$ and row index $R$ are calculated. The upper bound of the distance $U$ is the maximum distance in number of hops in the communication network, which is equal to the number of PEs in the worst case. The loop for a load is shown in Algorithm 5.2. For a load the loop always has $\#PE+1$ iteration as each data item has to reach the node that requests it. Though the load/store loop is not efficient, it allows mapping generic memory accesses on the proposed architecture, resulting in more flexibility.

**5.2.2 Interleaved Work-Item Mapping and Data Layout**

For generic kernels, the work-items are mapped onto the PE array linearly, i.e., the $i$-th work-item is mapped to the $i\%N$-th PE, where $N$ is the number of PEs. If there are multiple dimensions in the index space, only the first dimension is mapped to the vector, and the others are controlled by CP loops. In the PE array of the proposed architecture, each PE can only communicate with its left and right neighbors. Therefore long distance communication is not efficient. In this work, *interleaved* mapping and data layout are used to reduce the communication distance.

Figure 5.10 shows an example of the interleaved mapping. The index in Figure 5.10 represents both the work-item and the linear address of the data. Each work-item needs to access data in a window of size 5, e.g., work-item 4 needs to access data in addresses 2 to 6. If a linear mapping is used, each work-item needs to communicate with PEs two steps away, which
Algorithm 5.1: Generic PE store with communication

**Input**: Distance upper bound $U$, shift direction $d$, value vector $V$, destination PE ID $C$, destination row index $R$

1. # Loop through all PEs and check $C$ and $R$ dynamically.
2. # Distance upper bound can be determined by compiler.
3. for $i = 0$ to $U$ do
4.   for Each PE $i$ // Run in parallel on all PEs do
5.     if $C[i] == i$ then
6.       Store $V[i]$ in the memory PE[$i$][$R[i]$]
7.     end
8.   end
9. # The following can be skipped if compiler knows the
10. # values are the same for all PEs
11. Shift $V$ by 1 step in $d$ // Shift the value vector
12. Shift $C$ by 1 step in $d$ // Shift the destination PE index vector
13. Shift $R$ by 1 step in $d$ // Shift the destination row index vector
14. end

Algorithm 5.2: Generic PE load with communication

**Input**: Source PE ID $C$, source row index $R$

**Output**: Load value $V$

1. # Each PE needs a value from the memory bank of some PE.
2. # Loop through all PEs and check $C$ and $R$ dynamically.
3. # The loop has $\#PE + 1$ iterations each PE get the requested value.
4. for $i = 0$ to $\#PE$ do
5.   for Each PE $i$ // Run in parallel on all PEs do
6.     if $C[i] == i$ then
7.       Load memory PE[$i$][$R[i]$] to $V[i]$
8.     end
9.   end
10. Shift $V$ by 1 step in $d$ // Shift the load value vector
11. # The following can be skipped if compiler knows the
12. # values are the same for all PEs
13. Shift $C$ by 1 step in $d$ // Shift the source PE index vector
14. Shift $R$ by 1 step in $d$ // Shift the source row index vector
15. end
is less efficient in the proposed architecture. By using a mapping with interleaving factor of 2, as shown in Figure 5.10, the maximum communication distance is 1 instead of 2. When a kernel is mapped with an interleaving factor $N > 1$, the work-group loop has to be unrolled $N$ times in order to handle the irregular communication pattern, which may introduce energy overhead. For example, in Figure 5.10, work-item 4 needs two samples from left and one from right, while work-item 5 needs one from left and two from right. Therefore they need different instructions and the kernel has to be unrolled. Each OpenCL memory buffer object is analyzed by its access patterns. If there are different interleaving factors for multiple accesses (from the same or different kernels), the biggest one is used as the actual interleaving factor. The interleaving information is also used to generate the host code for proper data transfer between the system memory and the local memory of the accelerator. The host processor programs the DMA controller according to the interleaving factors determined by the device compiler. In the current implementation, kernels have to be compiled off-line in order to use the interleaved mapping.

The limitation of the interleaved mapping in this work is that it requires that: $i)$ all address expressions for the global and the local memory can be analyzed statically; $ii)$ the kernel launch parameters are compile-time constants.

### 5.2.3 Code Generation for Explicit Datapath

After mapping the kernels, the compiler treats the OpenCL kernel as a normal program and proceeds with the compilation. Most parts of the compiler back-end are similar to the compiler described in Section 3.3. The remainder of this subsection presents the parts that are specifically designed for the proposed wide SIMD architecture.

#### Legalizing Communication Code

In the proposed architecture, inter-PE communication and data exchange between CP and PE are constrained to the nearest neighbor. In addition, as described in Section 5.1.3, only the first source operand (Op A) can use the communication network (neighborhood and broadcast), otherwise the instruction contains illegal communication. Therefore a pass to check for instructions with illegal communication and transform them to legal communication is required. As shown in Figure 5.7, the legalization is done before the instruction scheduling. For a commutative operation with communication in the second operand, the compiler swaps the operands to make the communication legal. In other cases, the compiler inserts an extra copy instruction
for communication.

**Instruction Scheduling and Register Allocation**

The instruction scheduler is based on the one described in Section 3.3.1. A few modifications are required for the proposed architecture:

- The scheduler has to be aware of the resources for both CP and PE and manage them properly, including the separate issue slots.
- The generated schedule needs to meet the synchronization requirements in some communication between CP and PE. For example, in Figure 5.4, the CP instruction and the PE instruction have to be scheduled in the same cycle, in order for CP to get the correct value from the first PE.

The operand bypassing state is set by scanning the scheduled machine code. Register allocation is done using a graph-coloring based algorithm at function-level [20]. The RF of CP and the RF of PE are treated as two different register classes, and it is not allowed to directly copy from one class to the other. However, the schedule may change during register allocation if there is spilling. In that case, the operand bypass state initialization and the register allocation need to be run again. This process is illustrated in the loop in Figure 5.7. Though in practice the loop usually terminates quickly, because spill and reload codes on the proposed architecture do not need extra registers thanks to the explicit bypassing.

### 5.3 Experimental Results

In this chapter, area, performance and energy consumption are the metrics used to evaluate the proposed architecture. The kernels used for evaluation are listed in Table 5.2. The experimental setup is described in Section 5.3.1. Section 5.3.2 presents the area results. The performance and energy results are shown in Section 5.3.3 and Section 5.3.4, respectively.

#### 5.3.1 Experimental Setup

To evaluate the performance and energy efficiency of the proposed design, the kernels in Table 5.2 are tested on two types of processors: one with explicit bypassing, one with automatic bypassing. Three different setting are tested: 32 PEs, 64 PEs and 128 PEs. Other important

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAdd</td>
<td>Matrix addition</td>
</tr>
<tr>
<td>FIR</td>
<td>5-tap FIR filter on 1-D data stream</td>
</tr>
<tr>
<td>Sobel</td>
<td>3x3 Sobel edge detection filter</td>
</tr>
<tr>
<td>Transpose</td>
<td>Square matrix transpose</td>
</tr>
<tr>
<td>YUV2RGB</td>
<td>YUV to RGB color conversion</td>
</tr>
</tbody>
</table>
parameters of the two processors are listed in Table 5.3. All processors used in the experiments are implemented in synthesizable Verilog RTL. For a given processor instance, the RTL is generated automatically by the framework described in Chapter 6.

The kernel codes written in OpenCL are compiled by the proposed compiler off-line, i.e., at compile time. The generated binaries are executed on the cycle-accurate simulator for the SIMD processor to collect statistics. Due to the lack of implementation of a complete platform as shown in Figure 5.5, the host processor is emulated by a test driver program. Therefore, parts of the host overhead, e.g., the cost of re-organizing data layout (assuming the original data is organized in a non-interleaved fashion), are not reflected in the results in this section. Sequential codes written in C are compiled and run on a 4-stage 32-bit RISC processor as the reference (RISC). The RISC processor has 12kB instruction memory (24-bit) and 16kB data memory (32-bit).

### 5.3.2 Core Area

The core area of different processors used in the experiments is shown in Table 5.4. The results show that with the neighborhood network, the proposed architecture is scalable, as most part of the processor only contains local wires. Note that compared to cores with automatic bypassing, cores with explicit bypassing have smaller physical register files and simpler bypassing logic. As a result, the core area is reduced by 4.6%, 3.9%, 4.0% for 32-PE, 64-PE and 128-PE configurations, respectively.

### 5.3.3 Performance

Figure 5.11 shows the normalized cycle count of each kernel on different processors. The average speed-up of the proposed architecture with regard to the RISC reference is 27.24×, 48.70×, 85.59× for processors with 32-PE, 64-PE and 128-PE, respectively. The use of explicit bypassing does not have any visible impact on the performance. For kernels with regular memory access patterns, including the MAdd and FIR, the speed-up compared to RISC processor is ideal. For the FIR, Sobel and YUV2RGB kernels, the speed-up is larger than the number of PEs. This is because the proposed architecture is able to exploit the instruction-level parallelism by using two issue slots. It is also shown that by using the interleaved mapping described in Section 5.2.2, the FIR kernel gains extra 23.4% in performance for 128-PE.

<table>
<thead>
<tr>
<th>Table 5.3: Configuration of the target architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter</td>
</tr>
<tr>
<td>Data width</td>
</tr>
<tr>
<td>Pipeline stages</td>
</tr>
<tr>
<td>Number of PEs</td>
</tr>
<tr>
<td>Instruction memory</td>
</tr>
<tr>
<td>PE data memory</td>
</tr>
</tbody>
</table>
Table 5.4: Core area of different processors. The numbers inside parentheses are relative values. The unit of absolute values $\mu m^2$. The Cell area is the area occupied by logical cells, which is calculated using the technology library. The Net area is the estimated using the physical library information.

<table>
<thead>
<tr>
<th>#PE</th>
<th>RISC</th>
<th>Auto 64</th>
<th>128</th>
<th>Explicit 32</th>
<th>64</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10806 (1.0)</td>
<td>329952 (30.5)</td>
<td>645903 (59.8)</td>
<td>1275403 (118.0)</td>
<td>309597 (28.7)</td>
<td>607396 (56.2)</td>
</tr>
<tr>
<td></td>
<td>23411 (1.0)</td>
<td>747418 (31.9)</td>
<td>1454670 (62.1)</td>
<td>2894902 (123.7)</td>
<td>718526 (30.7)</td>
<td>1410852 (60.3)</td>
</tr>
<tr>
<td></td>
<td>34217 (1.0)</td>
<td>1077370 (31.5)</td>
<td>2100573 (61.4)</td>
<td>4170305 (121.9)</td>
<td>1028123 (30.0)</td>
<td>2018248 (59.0)</td>
</tr>
</tbody>
</table>

Figure 5.11: Speed-up compared to RISC

The performance of the matrix transpose kernel is considerably worse compared to other kernels. The main reason is that in matrix transpose, long distance communication is required. Since only neighborhood communication is possible, such a kernel is not efficient on the proposed architecture. Further software optimization is possible to improve the performance of such kernels on the proposed architecture [156].

5.3.4 Energy Consumption

In this work, the energy of both the core and memory are considered. The 7 processors (3 automatic bypassing SIMD + 3 explicit bypassing + 1 RISC) used in the experiments are synthesized using the TSMC 40nm low power library targeting 100MHz. The core energy consumption is accurately estimated using the physical information in the technology library and circuit toggle rate generated by post-synthesis simulation on the gate-level netlist. The energy consumption of the memory is estimated with CACTI [114]. Table 5.5 shows the energy consumption of
Table 5.5: Energy consumption of different data accesses

<table>
<thead>
<tr>
<th></th>
<th>4KB (32-bit)</th>
<th>14KB (56-bit)</th>
<th>16KB (32-bit)</th>
<th>12KB (24-bit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pJ/Access</td>
<td>2.02</td>
<td>3.37</td>
<td>2.92</td>
<td>2.49</td>
</tr>
</tbody>
</table>

different types of accesses.

To compare between scalar and vector register accesses, an access to the register file of one PE is considered equivalent to accessing the scalar register file. Figure 5.12 shows the number of register accesses. For the SIMD processor with automatic bypassing, the reduction in register file accesses comes mostly from the fact that the control-related instructions are run on the CP, which greatly reduces the number of register accesses in these instructions. It is clear that the explicit bypassing has dramatic impact on the number of register accesses. In particular, almost all RF accesses are eliminated in the MAdd and FIR. In the transpose kernel, the SIMD processor with automatic bypassing has much more register accesses than RISC due to the communication. In contrast, the processor with explicit bypassing is able to eliminate all register accesses in the communication, resulting in over 70% decrease in number of accesses.

Figure 5.13 shows the normalized energy results. Both SIMD processors benefit from the reduction in instruction memory, which is a natural advantage of SIMD architectures. Furthermore, it is also clear that the reduction in the register access has noticeable impact in the energy consumption. Compared to RISC, the explicit bypassing SIMD processors with 32-PE, 64-PE and 128-PE reduce the energy consumption by 51.6%, 49.9% and 49.5%, respectively. And the average energy improvement of using explicit bypassing over automatic bypassing is 26.6%, 30.6% and 33.2% for 32-PE, 64-PE and 128-PE respectively. As mentioned in Section 5.2.2, the interleaved mapping introduces a small overhead in the number of register access. The reason is that in the unrolled kernel, samples need to be stored in register for the following work-item, preventing the elimination of certain write-backs. But overall it still achieves energy reduction compared to the non-interleaved version due to cycle count reduction.

All in all, the proposed architecture and compiler are able to achieve substantial improvement in both performance and energy consumption. The proposed architecture is highly scalable in terms of area, performance and energy.

5.4 Related Work

Wide SIMD architectures are used in many embedded processors. The Xetal from NXP [1] is an SIMD processor with 320 PEs that is designed for smart camera data processing. The PEs in Xetal are connected by a neighborhood network. The PE data memory in Xetal can only be addressed globally, i.e., all PEs have to access the same row of the memory. Xetal was one of the most efficient processor at the time. However, due to the lack of a register file, the energy consumption of the vector memory is high. He et al. addressed this problem in Xetal-Pro, by introducing an extra level of memory, as well as aggressive voltage scaling, resulting in a
Figure 5.12: Number of register accesses (normalized to RISC)
Figure 5.13: Energy consumption (normalized to RISC)
much more efficient architecture [57]. The IMAPCAR from NEC [85] is another example of a wide SIMD processor. The IMAPCAR has 128 PEs connected with a ring network. A key difference in IMAP compared to Xetal is that IMAP has independent address generation for each PE. While the memory is more complex in such a configuration, it also results in much better programmability. The ClearSpeed CSX700 processor has two multi-threaded SIMD arrays, each containing 96 PEs [27]. Woh et al. proposed AnySP, a wide SIMD targeting wireless and multimedia applications [161]. The PE interconnect in AnySP is a reconfigurable RAM-based crossbar, which is more flexible compared to Xetal and IMAP. The energy of the vector register file in AnySP is reduced by introducing an extra 4-entry small register file. Wide SIMD architectures are also popular among GPUs [104, 79]. For example, in the NVIDIA Fermi architecture, a Streaming Multiprocessor (SM) contains 32 cores that operate in SIMD fashion [160]. Recently, mainstream general purpose processors are also starting to introduce wide SIMD processing. For example, Intel introduces Advanced Vector Extensions (AVX) that supports 256-bit wide SIMD processing in its CPUs [39]. In this work, the proposed architecture is similar to the Xetal-Pro [57]. The main difference is that the proposed architecture uses per-PE register file and index addressing, and a PE datapath with explicit bypassing. Experimental results show that the proposed architecture achieved high energy-efficiency.

Programming wide SIMD architectures has always been difficult. IMAP uses a dedicated C dialect called one-dimension C (1DC) to develop data-parallel processing programs [85]. The ClearSpeed CSX700 can be programmed using a proprietary C dialect named Cn, which introduces poly variables for data to be processed in the PEs. Languages like 1DC can be fine tuned for the target architecture and are used by similar architectures such as Xetal. But they lack portability and are not compatible with standard languages. GPU is one of the most common types of wide SIMD architectures. There has been a lot of effort in improving the programmability of GPUs, especially for general purpose computing on GPUs (GPGPU). NVIDIA introduced CUDA that enables programming GPUs for general purpose computing applications [91, 98]. OpenCL is a standard parallel programming language for heterogeneous platforms [116]. GPGPUs are among the most important target architectures of OpenCL [111, 160], but many different types of architectures can also efficiently support OpenCL. Karrenberg and Hack proposed methods to map OpenCL programs onto general purpose CPUs efficiently [74]. Recent studies attempt to use OpenCL for more diverse target architectures, such as FPGAs [103] and ASIPs [67]. In this work, in an effort to support standard programming language to low-energy wide SIMD architecture, we presented the design of a compiler that is capable of compiling OpenCL program on the proposed wide SIMD architecture.

5.5 Summary
In this chapter, an energy-efficient wide SIMD processor architecture is introduced. The proposed architecture uses an explicit datapath that dramatically reduce the accesses to the register file (RF) in the processing elements (PEs). A neighborhood-only communication network is used for communication between PEs, which makes the proposed architecture scalable. To im-
prove the programmability, an OpenCL compiler design is proposed for the target architecture. The compiler is able to analyze and map OpenCL kernels onto the proposed architecture. When the memory accesses in the kernels can be statically analyzed, the compiler is able to optimize the mapping of the data to improve the communication efficiency between PEs. Experimental results show that the proposed architecture is scalable and efficient. Compared to the RISC reference, a 128-PE instance achieves an average speed-up of $85 \times$, and reduces the average total energy consumption by 49.5%.

Currently we are working on further analysis and optimization of the memory layout and work-item scheduling. In particular, handling less static address expressions would be useful for generalizing the proposed architecture. It is also very interesting to perform software-hardware exploration for improving the inter-PE communication in an energy efficient way such that kernels with irregular communication pattern are easier and more efficient. In addition, further changes in the architecture, e.g., clustering PE memory banks to reduce memory energy, also require the adaption of the compiler.
In the previous chapters, we presented new architectures and compilation techniques that improve the energy efficiency of embedded processors for streaming applications. However, as discussed in Section 1.1, modern mobile devices rely on MPSoCs that contain various types of components. Therefore the energy efficiency of the whole system is determined not only by the efficiency of a processor itself, but also by how well it is integrated into the system. It is crucial to adapt to heterogeneity of the system when designing embedded processor architectures and tools. To achieve that, interfaces and tools that enable software-hardware co-design for emerging streaming applications are required. Commercial tools like the Synopsys Processor Designer [144] and Cadence Tensilica Customizable Processor IP [19] are good examples of design frameworks that aim to provide such capability.

Having an efficient interface, compiler and runtime environment that supports languages designed for heterogeneous system such as OpenCL is very important. In this chapter, a complete design framework is proposed. The framework is capable of automatically generating an implementation of accelerator IP based on the processor architecture proposed in previous chapters (particularly, in Chapter 5) for different target technologies, including ASIC and FPGA. The generated IP contains a standard AXI interface which enables efficient integration into MPSoCs. The software toolchain, including compiler, runtime libraries and simulator provides an efficient design environment for developing streaming applications on the proposed architecture within heterogeneous multi-core systems.

The remainder of this chapter proceeds as follows: Section 6.1 presents the automatic RTL generation in the proposed framework. The accelerator template based on the architecture proposed in the previous chapters is described in Section 6.2. The complete co-design framework is presented in Section 6.3, which contains tools for generating both software and hardware. In Section 6.4, we show examples of using the proposed framework. Finally, Section 6.5 concludes the findings in this chapter.
6.1 Automatic RTL Implementation Generator

The proposed framework is based on the wide SIMD processor architecture described in Chapter 5. Many parameters in the architecture are configurable, including:

- The number of pipeline stages in control processor (CP) and processing elements (PEs).
- The number of PEs in the PE array.
- Whether explicit bypassing is used in CP and/or PE.
- Other parameters of the datapath of CP and PE, such as the size of the register file (RF).

To efficiently implement processors with different parameters, we design and implement an automatic RTL generator. As depicted in Figure 6.1, the generator generates an implementation package of a processor based on a specification in a JSON file [66], including RTL codes and auxiliary files. The actual code generation is done using the Python template engine Jinja2 [129]. In addition to generating processor instances for specified parameters, the generator provides various options for different target technologies:

- RTL codes are optimized for different targets. For example, when targeting FPGAs, it is much more efficient to implement small memories like the register files in the look-up tables of the FPGA.
- For a Xilinx FPGA target, the generator generates system wrapper files, enabling the processor to be integrated as a peripheral in a multi-core processing system.
- For an ASIC target, a testbench that simulates the behavior of host and system interface is generated, along with auxiliary files for synthesizing and simulating the core in a standard ASIC design flow. Area, timing and power can be accurately estimated for different technology libraries.
The generated package contains auxiliary files for the specified target technology, which enable using the implementation in standard commercial toolflows directly.

### 6.2 Accelerator Architecture Template

The wide SIMD architecture proposed in Chapter 5 is highly efficient for streaming applications. However, it is not suitable for performing general purpose tasks, especially for control-intensive ones such as operating systems. Therefore we propose using the wide SIMD architecture introduced in Chapter 5 as an accelerator for streaming applications in a complete system. Figure 6.2 depicts a generic architectural template of such a system. The slave interface allows the host to have direct access to all the memories of the accelerator. For more efficient data transfer, a direct-memory-access (DMA) controller can be used to move data between external memory and the memories of the accelerator.

Figure 6.3 illustrates the basic execution model. By default the accelerator is in reset state. When there is a computing task, the host processor first configures the memories of the accelerator, including instructions and data. Then it clears the reset flag in the control register, which starts the execution of the wide SIMD accelerator. After finishing the computation task, the accelerator sets the status in the control register and goes back into reset state. The host gets the status update by either polling or interrupt, and retrieves the results from the data memory of the accelerator. The data copying can overlap with the computation when the memories in the accelerator accept simultaneous accesses from the interface and the SIMD processor (either multi-ported or with a bus). The process nicely fits in the execution model of OpenCL [116], making the proposed framework easy to program.

In the current framework implementation, the accelerator template is only used to generate an FPGA package. At the time of writing, we are exploring the design of a proper generic system interface for ASIC toolflows.

### 6.3 Hardware-Software Co-Design Framework

We propose a hardware-software co-design framework that enables efficient utilization of the proposed configurable architecture. Figure 6.4 shows the structure of the design flow. The architecture specification is given by the user in a JSON structured document. The specification file is used by three components in the framework: a compiler (described in previous chapters), an RTL generator (see Section 6.1) and a cycle-accurate simulator. The compiler is able to compile OpenCL programs for the specified processor (see Chapter 5). Basic runtime libraries for C and OpenCL programs are provided to the compiler back-end, which improves the programmability of the proposed platform.

---

1 In principle, the Control Processor (CP) is a generic scalar processor that can be used for general purpose processing. However it is not intended for complex and control-oriented tasks, and also lacks some essential features such as interrupt support.

2 As illustrated in Figure 6.4, only static linking is supported in the current implementation.
Figure 6.2: Accelerator interface and system memory mapping.

Figure 6.3: Accelerator execution flow.
Figure 6.4: Compiler and architecture implementation flow.
There are three ways to run the compiled programs on the proposed processor: i) run RTL simulation with the generated RTL code; ii) implement the RTL design on an FPGA and execute programs on it; iii) run the program on cycle-accurate simulator. The simulator behaves the same as the RTL simulation. Compared to RTL implementation, the cycle-accurate simulator has the following advantages:

- The simulation speed is much higher than RTL simulation.
- The same simulator can be used for different processors configurations. Whereas the RTL design has to be re-generated for FPGA, it is time consuming even for a relatively small design.
- It is able to collect statistics and detailed simulation trace for hardware and software analysis and optimization.
- It supports debugging for the software development, including: i) breakpoints; ii) runtime processor state (pipeline, RF and memory) observation; iii) interactive simulation.

As shown in Figure 6.4, all three means of program execution are able to produce information for further architecture and application optimizations. In particular, the adaption of standard ASIC design flow enables accurate power estimation, which is crucial for energy-efficient processor design.

The proposed framework enables hardware-software co-design for a streaming accelerator based on the architecture proposed in previous chapters. The generated accelerator implementation is easy to use within a standard toolflow for large multi-core system design.

### 6.4 Case Study

The proposed framework is able to generate wide SIMD processors based on flexible specification files. All the processors used in the experiments in Chapter 5 are generated by the proposed framework.

The ability to quickly generate processors with different configurations also enables design

---

**Figure 6.5:** Normalized EDP and EDAP of YUV2RGB in different configurations.
space exploration for systems with various constraints and different target applications. The experiments in Section 5.3 give an example of exploring different numbers of processing elements (PEs). The Energy-Delay-Product (EDP) of running YUV2RGB kernel on configurations with different number of PEs is depicted in Figure 6.5. The EDP is the product of normalized energy and delay (execution time), which is a metric that reflects both energy consumption and performance simultaneously. It is considered a good metric for evaluating the energy efficiency of a processor [45]. As shown by the EDP, the energy efficiency is improved as the number of PEs increases. The Energy-Delay-Area-Product (EDAP) in Figure 6.5 also shows that when area is taken into account, the proposed architecture still gets more efficient as the number of PEs increases. Most kernels used in Section 5.3 show similar characteristics as YUV2RGB. However, as illustrated in Figure 6.6, for matrix transpose, the proposed accelerator architecture has lower efficiency as the number of PEs increases. And if area is taken into account, the efficiency is actually lower than the RISC baseline.

With the proposed framework, various architecture parameters can be quickly explored for different applications, which helps in designing energy efficient architecture for a specific target application set.

### 6.5 Summary

In this chapter, we presented a hardware-software co-design framework that is able to generate implementations of accelerators for streaming application based on the processor architecture and the compiler proposed in previous chapters. The framework includes an automatic RTL generator that generates an implementation package of an accelerator based on the wide SIMD architecture described in Chapter 5. A compiler that can compile OpenCL program for the accelerator, and a simulator with debugging support are included in the framework. The framework is used for different applications, which demonstrates that it can be used to perform exploration in designing energy efficient processor for streaming applications.

There are still many possibilities to improve the framework. The accelerator interface can
be further improved to adapt the OpenCL programming model. The power estimation in the current framework relies on time-consuming gate-level simulation (a few orders of magnitude slower than cycle-accurate software simulation) and power analysis in ASIC toolflows. A fast and accurate power model would be helpful for early design space exploration.
Conclusions and Future Work

In this chapter, we conclude the findings in this thesis in Section 7.1. In Section 7.2, we give an overview of open questions and possible future research directions.

7.1 Conclusions

Low power is one of the most distinguishable features in an embedded system, as well as one of the biggest design challenges. Although dedicated Application Specific Integrated Circuits (ASICs) are highly efficient, their lack of flexibility means they cannot meet the requirement of the fast-changing demands of modern embedded devices. This thesis studies hardware and software methods to develop an energy efficient, programmable embedded processing platform that is able to bridge the gap between ASIC and processors. In this work, the compiler is considered to be in a central role in improving the energy efficiency of the processor. In this thesis, we presented research and engineering efforts in both architecture and code generation to overcome parts of the challenges in energy efficient design for streaming applications.

In Chapter 3, we looked into the register file (RF), which is one of the most frequently used, and most power-hungry components in a typical processor. Analysis reveals that in many applications, most variables are used locally only for a few times, resulting in a lot of redundant RF accesses. Explicit datapath architectures that enable fine-grained control in the software are good candidates for reducing RF accesses to these short-lived variables. In this work, we introduced an architecture that allows software to directly control the bypassing network. As the software has much more control over the datapath compared to conventional architectures, efficient code generation is a key to achieve high energy efficiency. We presented a compiler back-end for the proposed explicit bypassing architecture. The compiler includes algorithms to schedule instructions such that most of the unnecessary RF accesses are eliminated, while the performance is unaffected. Experimental results showed that over 70% of the RF accesses are eliminated, which leads to a core energy reduction of 15%, and a 9.19% reduction in total energy consumption is observed when memory is taken into account. Comparisons with processors built with the TTA Co-Design Environment (TCE) framework showed that the proposed method achieves better RF access reduction and has the same code density as a generic RISC
Chapter 4 presented a method to support flexible operation-pair patterns in a RISC-like processor with a compact 24-bit instruction set architecture (ISA). In this work, two problems are tackled: i) encoding a large number of special operation opcodes; ii) supplying sufficient data to the special function unit (SFU). Application analysis shows that operation pair patterns have good locality in many applications. Therefore we proposed a partially re-configurable instruction decoder that is able to support flexible operation pairs with only nine opcodes (for eight reconfigurable special instructions and one reconfiguration instruction). Explicit bypassing proposed in Chapter 3 is used to reduce the overhead of supplying more operands to the SFU. The efficiency of proposed method again relies on the compiler. We proposed a compiler back-end that selects operation pattern and generates efficient code. Comprehensive experiments were carried out. The results showed that the average dynamic instruction count is reduced by over 25%, and the total energy is reduced by an average of 15.8% compared to the RISC baseline. When high performance is required, the proposed architecture is able to achieve an average speed-up of $1.14 \times$ with 13.1% energy reduction compared to the baseline by introducing a multi-cycle SFU. The results demonstrated that the proposed solution achieved a good balance between the flexibility of special instructions and energy efficiency.

In Chapter 5, we proposed a scalable wide SIMD processor architecture and a compiler that supports OpenCL. The SIMD is based on the Xetal-Pro architecture. As the RF consumes an even larger portion of energy in SIMD architectures than in scalar architectures, the processing elements (PEs) in this work use the explicit datapath described in Chapter 3. Features that enable the mapping of programs written in the OpenCL parallel language were added to the proposed architecture. The design of a compiler that compiles OpenCL programs and optimizes memory mapping for the proposed architecture was presented. Detailed experiments were carried out on processors with different configurations. The results showed that the proposed architecture and compiler were able to achieve substantial improvements in both performance and energy consumption for OpenCL programs. In a 128-PE processor, the proposed architecture was able to achieve over 200 times speed-up and reduce the energy consumption by 49.5% compared to a basic RISC processor.

Finally, a complete hardware-software co-design framework for a configurable accelerator was presented in Chapter 6. The framework consists of an RTL generator, a compiler with runtime libraries, and a cycle-accurate simulator. The RTL generator generates implementations of an accelerator based on the wide SIMD architecture described in Chapter 5 for different target technologies, including ASIC and FPGA. The compiler can compile OpenCL programs for the accelerator. The cycle-accurate simulator with debugging support is able to perform fast simulation for architectures with different configurations. The proposed framework was used for different applications, which demonstrates that it can be used to perform exploration in designing energy efficient processor for streaming applications within a heterogeneous multi-core system.
7.2 Future Work

The previous chapters of this thesis proposed architecture and code generation solutions to various problems in designing energy efficient programmable embedded computing systems for streaming applications. There are still open issues that can be further studied.

- The explicit bypassing introduced in Chapter 3 is able to improve the energy efficiency of the RISC architecture. There are some opportunities to further improve the architecture and compiler:
  - In this work, all FU output and pipeline registers are exposed to the software by mapping to a register. However, this solution may not scale well for a more complex architecture with a larger number of FUs. Further exploration on how the intermediate results should be exposed is an interesting research topic.
  - The instruction scheduler for explicit bypassing introduced in Chapter 3 only operates at the basic block level. Extending it to a larger scheduling scope would be helpful for improving the efficiency in applications that are more control-intensive. Particularly, the control-flow join-point handling could be improved.
  - Supporting explicit bypassing in software-pipelining is quite useful for multiple-issue architecture like the one in Chapter 5.
  - Proper support for precise interrupt/exception handling is also an interesting topic. A method to support interrupt in a processor with explicit bypassing is proposed in [134]. However the method cannot be directly applied to the processor design in this work due to architectural differences. Further efforts are required to efficiently support interrupt in the proposed architecture.

- In Chapter 4, we proposed to support flexible operation pairs in a RISC processor. To further improve the energy efficiency, there are many possibilities:
  - Different types of special operation patterns could be explored, such as multiple-output patterns and patterns with more basic operations. As more complex patterns will lead to higher overhead, this requires careful exploration of the trade-offs between complexity and energy efficiency.
  - In the current compiler, the pattern selection is implemented as a separate pass before the instruction scheduling. More interaction between the pattern selector and other parts of the compiler, including the instruction scheduler and register allocator, could improve the quality of the pattern selection.

- We presented a wide SIMD architecture with neighborhood network and a compiler with OpenCL support for it in Chapter 5. Currently we are working on further analysis and optimization of the memory layout and work-item scheduling in OpenCL compilation:
  - In the current compiler, only address expressions that are fully statically analyzable can be handled efficiently. Further analysis and optimization techniques should be developed so the compiler could be more efficient for less static kernels.
In this work, the work-items are scheduled in a relatively straightforward way. More scheduling strategies can be explored to improve the performance and reduce the energy consumption. For example, by re-organizing work-items, it is possible to greatly reduce the amount of inter-PE communication.

The SIMD architecture itself also can be improved in several aspects:

- While the neighborhood network is scalable, some kernels with irregular communication patterns are difficult to map onto the proposed architecture efficiently. To tackle this problem, software-hardware exploration for improving the communication network has to be performed.
- The organization of the PE array, especially the vector memory layout can be explored. For example, in the current architecture, each PE has its own memory bank. Making several PEs sharing a bank could reduce the memory energy consumption. Such architecture changes also require the adaption of the compiler.
- The energy model used in this work mainly considers the dynamic energy consumption. With the shrinking feature size of semi-conductor technology, leakage energy is becoming more significant, especially for the vector memory. Therefore it is important to include that in the energy model and optimize for it. In [57, 122], evaluation and optimization techniques for the memory of wide SIMD are proposed. Similar method can be applied to the architecture in this thesis and achieve balance between dynamic and leakage energy.

- In Chapter 6, a co-design framework based on the proposed wide-SIMD architecture was presented. There are still many possibilities for improvement.
  - The accelerator interface can be further improved to adapt the OpenCL programming model, especially for the data transfer between external memory and the local memory.
  - The power estimation in the current framework relies on time-consuming gate-level simulation and power analysis in ASIC toolflows. A fast and accurate power model would be helpful for early design space exploration.

- The architecture and code generation techniques proposed in this thesis are tested using a set of kernels, many of which are relatively small. Though these are representative kernels in streaming application, it is also important to run large benchmark suites such as EEMBC [35], which can better evaluate the proposed methods and provide more information for further development.
References


References


REFERENCES


[81] Nam Sung Kim, Todd Austin, David Baaaw, Trevor Mudge, Krisztian Flautner, Jie S. Hu, Mary Jane Irwin, Mahmut Kandemir, and Vijaykrishnan Narayanan. “Leakage current: Moore’s law meets static power”. In: *Computer* 36.12 (2003), pp. 68–75.


[113] Delft University of Technology. MOVE. URL: http://ce.et.tudelft.nl/MOVE/.


In this appendix, we introduce the RISC Instruction Set Architecture (ISA) that is used as the baseline ISA in this thesis. The ISA is based on the OpenRISC ISA [102]. One key difference from the OpenRISC ISA is that the instruction width is reduced from 32 bits to 24 bits. Similar to ARM Thumb [112] and MIPS16 [115], the more compact instruction set can lead to higher efficiency for embedded processors. Unlike ARM Thumb and MIPS16, the proposed compact ISA still have access to the whole architectural register file. The main difference is in the size of the immediate and the number of opcodes it can support, and in this thesis, interrupt or exception is not considered.

The remainder of this appendix proceeds as follows. Section A.1 gives an overview of the baseline ISA. Section A.2 introduces the registers in the baseline ISA. The three different instruction formats are described in Section A.3. Section A.4 specifies different types of supported operations. Section A.5 summaries the baseline ISA.

### A.1 Instruction Set Architecture Overview

Table A.1 lists the key parameters of the compact 24-bit baseline Instruction Set Architecture (ISA). Similar to other RISC architectures, the baseline ISA is a load-store architecture, that is, only load and store instructions access the data memory, while other instruction only uses the register file (RF) for storing temporary data. For the research purpose in this thesis, only integer operations are included. For all binary operations, the baseline ISA uses a three-address instruction format: each instruction specifies the addresses for the destination of the result, the

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native data width</td>
<td>32 bits</td>
</tr>
<tr>
<td>Instruction width</td>
<td>24 bits</td>
</tr>
<tr>
<td>Data address width</td>
<td>32 bits</td>
</tr>
<tr>
<td>Logical address space</td>
<td>One linear address space</td>
</tr>
</tbody>
</table>
first source operand, and the second source operand. In the assembly language, an operation that computes \( d = a \ OP \ b \) is denoted as:

\[
\OP d, \ a, \ b
\]

where \( \OP \) is the mnemonic of the operation, \( d \) is the destination, and \( a, b \) are the first and the second source operands, respectively.

### A.2 Registers

In the baseline ISA, there is a architectural Register File (RF) with 32 registers, two read ports and one write port. As mentioned in A.1, all operations in the baseline ISA except load/store use registers for temporary storage. Therefore the RF is a central component. The list of RF registers is given in Table A.2. The register R0 always has the value zero, and any write to it is discarded. The register R9 is the link address register. When a jump-and-link (see Section A.4.1) instruction is executed, the value of the current program counter is automatically stored in R9 as the return address. The special behavior of R0 and R9 is enforced by the hardware. Therefore the use of these registers in the software is restricted, for example, R9 can be used safely only if there is no jump-and-link (function call) instructions between the live range of the value stored in it. Other registers in the RF can be used by the software for any purpose.

In addition to the RF, there are two special registers in the baseline RF, PC and F. PC is the program counter register, which contains the address of the current instruction. Some branch operations implicitly use PC as a source operand (see A.4.1). The value of PC is automatically incremented every cycle, or is set by a branch operation. F is the flag register, which holds a 1-bit value and is set by compare operations (see A.4.2). F is used as a source operand in conditional move operations (see A.4.2) and conditional branch operations (see A.4.1).

### A.3 Instruction Format

There are three instruction formats in the baseline ISA, namely, J-Type, R-Type and I-Type. The J-Type is used for control-related operations. The R-Type and I-Type are used for binary operation and load/store operations. The remainder of this section gives a detailed description of each instruction format.

<table>
<thead>
<tr>
<th>Register</th>
<th>Usage</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>Always zero</td>
<td>Any write to R0 is ignored</td>
</tr>
<tr>
<td>R1 - R8</td>
<td>For general purpose</td>
<td></td>
</tr>
<tr>
<td>R9</td>
<td>Link address register</td>
<td>Value is maintained by hardware</td>
</tr>
<tr>
<td>R10 - R31</td>
<td>For general purpose</td>
<td></td>
</tr>
</tbody>
</table>
Figure A.1: J-Type instruction formats. There are two different formats, the top one has a immediate operand and the bottom one has a register operand.

Figure A.2: R-Type instruction format.

A.3.1 J-Type

The J-Type instructions are for control-related operations, including branch operations (Section A.4.1) and miscellaneous operations (Section A.4.5). Figure A.1 illustrates the two different formats of J-Type instructions, one with 16-bit immediate operand, the other with one register operand. The immediate operand is sign extended to a 32-bit value.

A.3.2 R-Type

The R-Type instructions are for operations that uses registers as input operands. Figure A.2 depicts the format of R-Type instructions. Note that to avoid ambiguity with J-Type instructions, the opcode 0 cannot be use for R-Type instructions. In an R-Type instruction for a binary operation, Rd is the RF index of the destination, Rs is the RF index of the first source operand and Rt is the RF index of the second source operand. The first three bits (bit 0-2) are reserved for future use.

A.3.3 I-Type

The I-Type instructions are for operations that uses both registers and immediate values as input operands. Figure A.3 illustrates the format of I-Type instructions. Depending on the operation, the 8-bit immediate operand IMM is sign or zero extended to 32-bit in the operations.

Immediate Type

This is a special type of instructions that used for specifying long immediate values for I-Type and J-Type instructions. The semantics of this type of instructions is explained in Section A.4.4.
Table A.3: List of branch operations. The immediate operand IMM is sign extended.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Semantics</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>BF</td>
<td>Branch to PC + IMM if F == 1</td>
<td>Immediate operand</td>
</tr>
<tr>
<td>BNF</td>
<td>Branch to PC + IMM if F == 0</td>
<td>Immediate operand</td>
</tr>
<tr>
<td>J</td>
<td>Branch to PC + IMM</td>
<td>Immediate operand</td>
</tr>
<tr>
<td>JR</td>
<td>Branch to Rt</td>
<td>Register operand</td>
</tr>
<tr>
<td>JAL</td>
<td>Jump to PC + IMM and store PC to R9</td>
<td>Immediate operand</td>
</tr>
<tr>
<td>JALR</td>
<td>Jump to Rt and store PC to R9</td>
<td>Register operand</td>
</tr>
</tbody>
</table>

A.4 Supported Operations

In thesis, the baseline ISA supports basic integer operations. The operation set of the baseline ISA is comparable to the integer operations in typical RISC ISAs like MIPS [115], thereby providing a fair baseline for the research in this thesis. The remainder of this section gives a detailed description of the operations in the baseline ISA. Section A.4.1 describes the branch operations. Section A.4.2 introduces the integer arithmetic and logical operations. Memory operations are described Section A.4.3. The immediate instructions that used to load long immediate values for I-Type instructions are presented Section A.4.4. Finally, Section A.4.5 describes miscellaneous operations.

A.4.1 Branch Operations

Table A.3 shows the branch operations in the baseline ISA. The conditional branch operations uses the flag register F as the source operand. For branch operations with immediate operands, the target address is calculated as PC + IMM, resulting in a branching range of 64KB. The support of jump-and-link (JAL, JALR) enables efficient implementation of functions calls.

All branch operations have one delay slot, that is, the instruction that immediately follows a branch operation is always executed. The branch delay slot enables zero branch delay in a pipelined implementation of the ISA.

A.4.2 Integer Arithmetic and Logical Operations

Table A.4 lists the integer arithmetic and logical operations supported in the baseline ISA. Both R-Type and I-Type instructions are used to encode arithmetic and logical operations. For compare operations, the destination is always the flag register F, therefore the destination index Rd is always 0. For other arithmetic and logical operations, the destination is always in the register file.

Note that all operations except SUB and RSUB have identical semantics in I-Type and R-Type instructions. The reason is that the I-Type version of SUB can be replace by an ADD with immediate operand. Therefore the opcode of SUB is used to encode the SUB with reversed operand order, namely, RSUB, which cannot be replaced by an ADD with immediate operand.
Table A.4: List of integer arithmetic and logical operations. In the operation semantics, OpB is the second source operand, which is Rt for R-Type instructions, and IMM for I-Type instructions is sign extended unless specified otherwise.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Semantics</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>Rd = Rs + OpB</td>
<td></td>
</tr>
<tr>
<td>SUB</td>
<td>Rd = Rs - Rt</td>
<td>Only R-Type</td>
</tr>
<tr>
<td>RSUB</td>
<td>Rd = IMM - Rs</td>
<td>Only I-Type, same opcode as SUB</td>
</tr>
<tr>
<td>MUL</td>
<td>Rd = Rs * OpB</td>
<td></td>
</tr>
<tr>
<td>MULU</td>
<td>Rd = Rs * OpB</td>
<td>Unsigned MUL, IMM zero extended</td>
</tr>
<tr>
<td>OR</td>
<td>Rd = Rs</td>
<td>OpB</td>
</tr>
<tr>
<td>AND</td>
<td>Rd = Rs &amp; OpB</td>
<td>IMM zero extended</td>
</tr>
<tr>
<td>XOR</td>
<td>Rd = Rs ⊕ OpB</td>
<td>IMM zero extended</td>
</tr>
<tr>
<td>CMOV</td>
<td>Rd = (F == 1) ? Rs : OpB</td>
<td></td>
</tr>
<tr>
<td>EQ</td>
<td>F = (Rs == OpB)</td>
<td></td>
</tr>
<tr>
<td>NE</td>
<td>F = (Rs != OpB)</td>
<td></td>
</tr>
<tr>
<td>LE</td>
<td>F = (Rs ≤ OpB)</td>
<td></td>
</tr>
<tr>
<td>LT</td>
<td>F = (Rs &lt; OpB)</td>
<td></td>
</tr>
<tr>
<td>GE</td>
<td>F = (Rs ≥ OpB)</td>
<td></td>
</tr>
<tr>
<td>GT</td>
<td>F = (Rs &gt; OpB)</td>
<td></td>
</tr>
<tr>
<td>LEU</td>
<td>F = (Rs ≤ OpB)</td>
<td>IMM zero extended</td>
</tr>
<tr>
<td>LTU</td>
<td>F = (Rs &lt; OpB)</td>
<td>IMM zero extended</td>
</tr>
<tr>
<td>GEU</td>
<td>F = (Rs ≥ OpB)</td>
<td>IMM zero extended</td>
</tr>
<tr>
<td>GTU</td>
<td>F = (Rs &gt; OpB)</td>
<td>IMM zero extended</td>
</tr>
<tr>
<td>SLL</td>
<td>Rd = (Rs ≪ OpB)</td>
<td></td>
</tr>
<tr>
<td>SRA</td>
<td>Rd = (Rs ≫ OpB)</td>
<td>Shift right arithmetic</td>
</tr>
<tr>
<td>SRL</td>
<td>Rd = (Rs ≫ OpB)</td>
<td>Shift right logical</td>
</tr>
</tbody>
</table>
Table A.5: List of memory operations. Only aligned accesses are supported. In all operations, IMM is sign extended. For store operations, the data item to be stored in Rs.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Operation</th>
<th>Effective Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>SW</td>
<td>Store single word (32-bit)</td>
<td>Rd + (IMM \ll 2)</td>
</tr>
<tr>
<td>SH</td>
<td>Store half word (16-bit)</td>
<td>Rd + (IMM \ll 1)</td>
</tr>
<tr>
<td>SB</td>
<td>Store single byte (8-bit)</td>
<td>Rd + IMM</td>
</tr>
<tr>
<td>LW</td>
<td>Load single word (32-bit)</td>
<td>Rs + (IMM \ll 2)</td>
</tr>
<tr>
<td>LH</td>
<td>Load half word (16-bit)</td>
<td>Rs + (IMM \ll 1)</td>
</tr>
<tr>
<td>LB</td>
<td>Load single byte (8-bit)</td>
<td>Rs + IMM</td>
</tr>
</tbody>
</table>

Table A.6: List of immediate operations. Both operations are encoded as I-Type instructions, but all operand bits ([17:0]) are treated as the immediate value.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZIMM</td>
<td>Load immediate bits with zero extension</td>
</tr>
<tr>
<td>SIMM</td>
<td>Load immediate bits with sign extension</td>
</tr>
</tbody>
</table>

### A.4.3 Memory Operations

In the baseline ISA, memory operations uses basic offset addressing mode. Only I-Type instruction can execute memory operations. The effective address of an memory operations is the content of a base register plus an immediate offset. The list of memory operations in the baseline ISA is given in Table A.5. All memory operations only access aligned addresses, that is, the lowest two bits in the effective address of a single word access (LW or SW), and the lowest bit in the effective address of a half word access (LH or SH) are discarded. The data loaded by LH and LB is zero extended in the 32-bit register. In SH and SB, the lower bits of the data in Rs are stored in the memory.

### A.4.4 Immediate Operations

In the baseline ISA, an I-Type instruction can only use an 8-bit immediate value, which has relatively limited range (0 to 255 for an unsigned integer and \(-128\) to \(127\) for a signed integer). To enable I-Type instruction to use long immediate values, the baseline ISA introduces two special immediate operations ZIMM and SIMM, as shown in Table A.6. An immediate instruction does not perform any useful operation by itself. It is used to load the higher 24 bits of the immediate value in the I-Type instruction that immediately follows it. For example, in the following sequence two instructions:

```
SIMM 3
ADD R1, R2, 1
```

The value of the second operand of the ADD instruction is \((3 \ll 8)|1 = 769\). Note that when an immediate instruction proceeds an I-Type instruction, the 8-bit immediate operand in the I-
### Table A.7: List of miscellaneous operations.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOP</td>
<td>No operation</td>
</tr>
<tr>
<td>SYS</td>
<td>System call, with IMM as argument</td>
</tr>
</tbody>
</table>

Type instruction is not extended, but is only used as the lower bits of the final immediate value. Immediate instructions enable immediate operands of up to 26 bits in an I-Type instruction at the cost of one extra instruction.

#### A.4.5 Miscellaneous Operations

Table A.7 shows the two miscellaneous operations in the baseline ISA. NOP is useful for filling the cycles that cannot be used by any useful operations but is required (e.g., the unfilled branch delay slot). SYS is needed for implementing operating system kernel functionality.

#### A.5 Summary

In this appendix, we presented a RISC Instruction Set Architecture (ISA) that is used as the baseline architecture in this thesis. The baseline ISA is based on the OpenRISC ISA. The instruction width is reduced to 24-bit by reducing the operation set and immediate size, resulting in a more compact ISA. The operation set of the baseline ISA contains essential integer operations, which are sufficient for most applications. All in all, the ISA described in this appendix is a representative compact RISC ISA, and it is a fair baseline for the research in this thesis.
Finally, I have reached the time to write the last section of my PhD thesis. The past five years have been a tough but exciting journey. I would not have made it without the help and support of the kind people around me, and I would like to take this opportunity to express my gratitude.

First and foremost, I would like to thank my supervisor prof.dr. Henk Corporaal. He offered me the opportunity to pursue my PhD. He also has given me guidance, as well as freedom to exploit research directions. His insightful comments and sharp questions have been vital for this research. I am grateful for having the chance to work under his supervision. I would like to also thank my copromotor dr. Marcel Beemster. Even though he was only involved in the last year, his great feedback and advice have improved the quality of this thesis considerably.

I also would like to thank dr. Cristina Silvano, prof.dr.ir. Henk Sips, prof.dr. Kees Goossens, prof.dr.ir. Kees van Berkel and dr. Andrei Terechko for taking place in my doctorate committee. I also thank them for the valuable feedback to improve this thesis.

I would like to thank all others that contributed to this research. I am particularly grateful to Yifan He, for our numerous discussions and co-operations that lead to fruitful outcomes. I thank Zhenyu Ye, Luc Waeijen, Sander Stuijk, Sebastian Moreno Londono, Ashsan Shabbir and Bart Mesman for collaborating in research works. I also thank the members of the PARsE team, including Cedric Nugteren, Gert-Jan van den Braak, Maurice Peemen, Shakith Fernando, Roel Jordans, Erkan Diken, Rosilde Corvino and all the others, for their knowledge sharing and constructive discussions during all the PARsE meetings.

I also would like to thank everyone at the Electronic Systems group. My thanks go to my office-mates Marcel Steine, Morteza Damavandpeyma, Maurice Peemen, Gert-Jan van den Braak, Mathias Funk, Gabriela Breaban and Rachana Kumar. I would also like to thank the group leaders Ralph Otten and Twan Basten, and the secretaries Marja de Mol-Regels, Rian van Gaalen and Margot Gordon, for all their work and support.

I would like to thank my friends who made my PhD life in the Netherlands amazing. I am grateful to Yifan He and his wife Songyue Chen, for the great help they have provided in the past years. I thank Yang Yang for so many inspiring discussions. I thank Bo Liu for helping
me in so many ways, including picking my first car. Thanks also go to Yonghui Li, Zhenyu Ye, Zhengjie Lu, Wei Tong, Hubiao Yang, Rendong He, Pu Li, and many many others.

Last but not least, I would like to thank my parents Zhiguang She and Hui’e She, my wife Pinxiang Duan, and my parents-in-law Zhonghe Duan and Xiufang Guo, for all their unconditional love and support.

Dongrui She
October, 2014
Dongrui She was born in Guangdong, China, on January 24, 1985. He received the Bachelor degree in System Science and Engineering in 2007 from Zhejiang University (ZJU), China. In 2009, he received the M.Sc. degree in Embedded Systems from Eindhoven University of Technology (TU/e), The Netherlands.

In 2009, he started working towards a PhD degree within the Electronic Systems group at the department of Electrical Engineering of Eindhoven University of Technology. His work was funded by the Dutch Technology Foundation STW, under the Netherlands Streaming (NEST) project. His research focused on energy efficient architecture and code generation techniques for streaming applications. The work has led, among others, to several publications and this thesis.

Dongrui is currently a compiler engineer at Intel, Eindhoven, The Netherlands.
List of Publications

Refereed Journal Papers


Refereed Conference and Workshop Papers


