OUKA
Osaka University Knowledge Archive

Title	Low Power Design Method for Embedded Systems using VLIW Processor
Author(s)	小林, 悠記
Citation	大阪大学, 2007, 博士論文
Version Type	VoR
URL	https://hdl.handle.net/11094/1631
rights	
Note	

Osaka University Knowledge Archive : OUKA

https://ir.library.osaka-u.ac.jp/

Osaka University

Low Power Design Method
for Embedded Systems using VLIW Processor

July 2007

Yuki KOBAYASHI

Low Power Design Method for Embedded Systems using VLIW Processor

Submitted to
Graduate School of Information Science and Technology
Osaka University

July 2007

Yuki KOBAYASHI

Publications

Journal Articles (Refereed)

[J1] Yuki Kobayashi, Shinsuke Kobayashi, Keishi Sakanushi, Yoshinori Takeuchi, and Masaharu Imai: “HDL Generation Method for Configurable VLIW processor,” IPSJ Journal, vol. 45, no. 5, pp. 1311–1321, May, 2004 (in Japanese).

[J2] Yuki Kobayashi, Murali Jayapala, Praveen Raghavan, Francky Catthoor, and Masaharu Imai: “Methodology for Operation Shuffling and L0 Cluster Generation for Low Energy Heterogeneous VLIW Processors,” ACM Trans. on Design Automation of Electronic Systems (to appear).

International Conference Papers (Refereed)

[I1] Yuki Kobayashi, Shinsuke Kobayashi, Koji Okuda, Keishi Sakanushi, Yoshinori Takeuchi, and Masaharu Imai: “Synthesizable HDL Generation Method for Configurable VLIW Processors,” in Proc. Asia and South Pacific Design Automation Conference (ASP-DAC), pp. 843–846, Jan., 2004.

[I2] Yuki Kobayashi, Murali Jayapala, Praveen Raghavan, Francky Catthoor, and Masaharu Imai: “Operation Shuffling for Low Energy L0 Cluster Generation on Heterogeneous VLIW Processors,” in Proc. Embedded Systems for Real-Time Multimedia (ESTIMedia), pp. 81–86, Sep., 2005.

Domestic Conference Paper

[D1] Yuki Kobayashi, Shinsuke Kobayashi, Toshiyuki Sasaki, Koji Okuda, Keishi Sakanushi, Yoshinori Takeuchi, and Masaharu Imai: “Synthesizable HDL Generation Method for Configurable VLIW Processors,” IPSJ Symposium Series, vol. 2003, no. 11, pp. 259–264, Jul., 2003 (in Japanese).

Summary

Nowadays, embedded systems require severe design constraints, such as high performance and low energy or low power consumption. Design productivity is also a major factor for designing embedded systems since life time of industrial products is becoming shorter and shorter, while designing complex microprocessors needs design space exploration, where designers have to try designing and evaluating a lot of architecture candidates.

Superscalar processors are well known as high performance processor architecture that issue multiple operations. However, superscalar processors need a special hardware to extract the parallelism from the instruction stream, and the additional hardware increases power consumption. To achieve high performance in embedded systems where low power is also a key factor, VLIW processors is assumed as a reasonable solution since they can issue multiple operations at a time and a compiler extracts the parallelism; without a special hardware, the power consumption of VLIW processors is smaller than that of superscalar processors.

Although VLIW processors are effective solution for embedded systems which require both of high performance and low power, there are a lot of architectural parameters to be decided by designers. Since these parameters significantly affect the performance and area, it is required to perform the design space exploration where designers evaluate many architectures to determine the optimal parameter set. However, designing a VLIW processor is very complex, and consequently time consuming and error-prone. Hence, design space exploration on VLIW processors could not have been performed sufficiently so far.

Chapter 3 describes a synthesizable HDL generation method for configurable VLIW processors, which supports a flexible architecture model, especially in dispatching rules. Experimental results shows that the proposed method can generate a VLIW processor from a high level specification description, which is 80% to 90% smaller than HDL description. And also, the generation time of HDL description is sufficiently short, that is from 2 to 15 seconds. Since the specification description supports a wide range of dispatching rules and the amount of description is sufficiently small, it is possible to generate a wide range of fine-quality VLIW processors in a short time. Therefore, the proposed method can significantly improve the design productivity of VLIW processors. *(Related publications: [1] and [2])*

Then this thesis comes to the challenge for low power design. The power breakdown of VLIW processors indicates that the power bottleneck of VLIW processors is in the instruction memory hierarchy (e.g. instruction fetch). A loop buffer or L0 buffer architecture has been proposed to reduce the energy in the instruction memory hierarchy. The buffer locates between the instruction cache and the processor core and stores frequently-executed code to reduce the buffer access to the more power-consuming instruction cache. For further energy efficiency of L0 buffer in VLIW processors, L0 cluster architecture has also been proposed. Since the

architecture controls the buffer access more efficiently, the energy consumption is reduced furthermore. The result of L0 cluster generation is, however, sensitive to the schedule of target application.

An operation shuffling algorithm is described in Chapter 4. This algorithm improves energy efficiency of L0 cluster by changing operation scheduling. Since an L0 cluster configuration is very sensitive to operation scheduling, various schedules are generated and evaluated in order to obtain an optimal schedule. By shuffling all basic blocks iteratively, energy consumption can be reduced significantly. To reduce the size of the exploration space, some heuristics are also described in Chapter 4. The experimental results show that the proposed operation shuffling algorithm successfully reduces the energy consumption in various VLIW processors including heterogeneous VLIW processors as well as homogeneous VLIW processors. *(Related publications: [3] and [4])*

Since the simple operation shuffling takes huge amount of time even if the above heuristics are applied, a more efficient method to find a low energy operation schedule is then described in Chapter 5. Based on the analysis of characteristics of energy efficient L0 cluster configuration obtained from the operation shuffling, it is found that the optimal L0 cluster configuration is fixed after the first iteration of operation shuffling. Therefore, in the proposed method, the operation shuffling is performed only once for the most significant basic block and a compiler schedules again for the obtained cluster configuration. Some algorithms to schedule for a given cluster configuration are described in Chapter 5. By exploiting the scheduling algorithms, a compiler can generate a low energy schedule in a straightforward way. The experimental result shows that the proposed method can generate energy efficient schedules with 50 times shorter exploration time.

Preface

Embedded systems are widely used in our daily life. People use the embedded systems unawarely, but most of processors in the world are embedded into such a system. Therefore, working on the field of embedded systems is very exciting for me.

Nowadays, the life cycle of industrial products is going to be shorter and shorter. One of this reasons is globalization which is typically led by growth of the internet. Merits and demerits of globalization aside, people wants to have a new product sooner and people involved in the development has to design a new product sooner.

Design automation, or electronic design automation (EDA), is a representative technology in this era. We cannot ignore the technology growth and have to keep up with the technology.

Low power or low energy is becoming a keyword in recently years. Global warming cannot be ignored and it will be a main bottleneck of human activity in near future.

Here it is very important for us to pursue a method which manages these challenges of embedded systems, that are design productivity and energy consumption.

This thesis first describes a generation method for configurable VLIW processors. This method enables high design productivity and make the design space exploration easier.

An operation shuffling algorithm is then described in this thesis. This algorithm improves energy efficiency by changing operation scheduling. Since an L0 cluster configuration is very sensitive to operation scheduling, various schedules should be evaluated in order to obtain an optimal schedule. By shuffling all basic blocks iteratively, energy consumption can be reduced significantly. To reduce the size of exploration space, some heuristics are also described.

Since a simple operation shuffling takes huge amount of time even if the above heuristics are applied, a more efficient method to find a low energy operation schedule is then described. By exploiting some scheduling algorithms, a compiler can generate a low energy schedule in a straightforward way.

I hope this thesis will make a significant direction in this field for our future.

Acknowledgements

I would like to deeply thank Prof. Masaharu Imai, Osaka University, my supervisor. He has continuously supported me and my research and I always appreciate his technical insight which is always on target. Without his graciousness, I would not have enjoyed this fruitful period in my PhD days.

I would also like to thank Prof. Francky Catthoor, IMEC vzw and Katholieke Universiteit Leuven. He kindly welcomed me to IMEC and gave me a lot of thoughtful and valuable comments on my research work including journal and conference papers, as well as this thesis.

I am thankful to Prof. Takao Onoye, Osaka University, for reviewing this thesis.

I would like to thank associate professor Yoshinori Takeuchi and assistant professor Keishi Sakanushi for their daily advice. They have always taken care of our research and other worries.

I am thankful to Dr. Murali Jayapala, IMEC vzw. Without him, I would not have finished my work. He had made a basis of the work described in the latter half of this thesis. His comments always made me encouraged.

I thank Hiroaki Tanaka, Ittetsu Taniguchi, Takashi Hamabe, Hirofumi Iwato, Takuji Hieda, and other all members of Integrated System Design Laboratory in Osaka University for their helpful comments and suggestions in many aspects, especially in weekly seminars, and so on. I also thank Dr. Kyoko Ueda and Dr. Mohamed AbdElSalam Hassan. Special thanks to my contemporaries, Noboru Yoneoka, Hiroaki Tanaka, and Tatsuhiro Yoshimura. Conversation, lunch, and business trips with them were very pleasant, and have influenced my research as well. Chatting with friends makes me relaxed. I also thank secretary Yukako Nishikawa for her help especially on administrative papers.

I wish to thank Dr. Makiko Itoh, who made a basis of the work described in the former half of this thesis, and I would like to thank Dr. Shinsuke Kobayashi, who also helped the work a lot especially for the first few years in my research.

I wish to thank Andy Lambrechts, Praveen Raghavan. They helped me a lot in the beautiful country, Belgium. I also thank Daniele Scarpazza, Estela Rey Ramos, Javed Absar, David Novo Bruna, Theo Marescaux, Tom Vander Aa, Will Moffat, and other friends in IMEC. The stay in IMEC definitely has a significant influence not only for the period but also on whole of my life.

Finally, I owe a great deal of thanks to Yukiko who gave of her tenderness and sympathy during my PhD days and I would like to thank my parents and brother for supporting me through the years.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Embedded systems are widely used in our daily life and almost all of them have a programmable microprocessor inside. People might think most of microprocessors in the world are in personal computers; we call such kind of processor a general purpose processor (GPP). The number of GPPs inside personal computers is, however, less than 2% of the number of total processors in the world and most of processors locate in embedded systems [5]. An embedded system is usually dedicated to a few specific tasks, while a personal computer is intended to a wide range of applications. A portable audio player, mobile phone, and automobile are typical examples of embedded systems. We often see automated teller machines or electronic billboards in the city, most of which are controlled by microprocessors embedded. Thus, these kinds of embedded system have become an essential part of human activities in these days. Therefore, managing problems concerning microprocessors in embedded systems is very significant and profitable study.

Embedded systems often require severe constraints on performance and energy consumption. For instance, a modern audio player or mobile phone has a functionality of video player, as well as audio processing. Since video encoding or decoding needs huger computational effort than audio processing, such a system requires much more performance than ten years ago when the main task of such a system is only a simple audio processing. At the same time, such portable devices are typically battery-powered and they usually have limited space for a battery. Hence, they require extremely low energy consumption.

An embedded system can be typically implemented by using ASICs (Application Specific Integrated Circuits), GPPs, or ASIPs (Application Specific Instruction-set Processors). The advantage of ASICs is in the higher performance per area and lower power consumption than GPPs, but the limitation of ASICs is in the flexibility and extensibility in terms of a change of specification after the design completion. On the other hand, GPPs have the flexibility as a functionality of system is implemented by software which is programmable. GPPs are, however, usually not optimal for a target application. Hence, sometimes GPPs do not meet a performance requirement or they often exceed a power limitation of embedded systems. ASIPs can fulfill the requirements for flexibility as they are programmable using software, and for performance as they have an application specific instruction set. Therefore, ASIPs are appropriate for embedded systems due to flexibility and performance, as shown in Fig. 1.1.

Configurable processors are often used for designing ASIPs. Configurable processors have

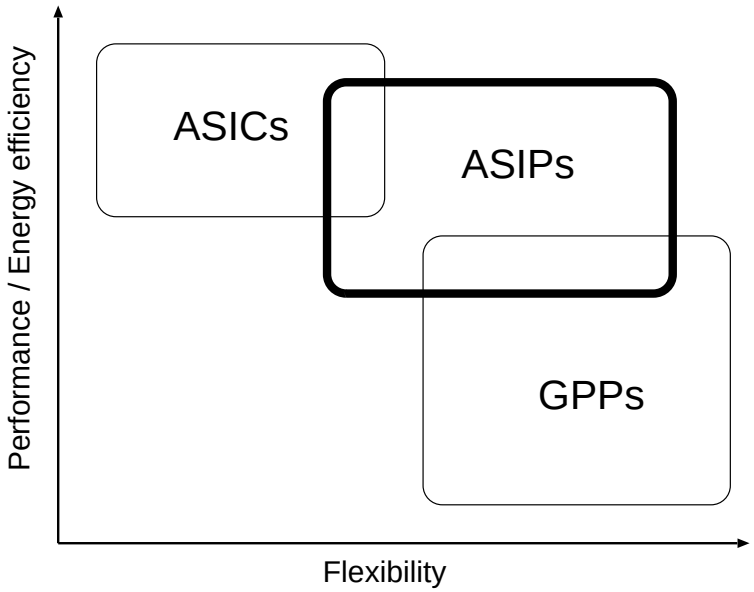


Figure 1.1: Advantage of ASIPs.

some parameters to tune up their instruction set, such as the bit width of data path, the number of general purpose registers, and additional instructions. By using configurable processors, the design time can be shorter than the manual design since the basic architecture is almost fixed. While configurable processors can be more general purpose oriented, it can be more energy efficient or higher performance when their instruction set is tuned. Therefore, configurable processors are reasonable solution for the design of ASIPs.

1.1 VLIW processor

In order to realize a higher-performance instruction set processor, superscalar architecture [6] and VLIW (Very Long Instruction Word) architecture [7] are proposed, which exploit instruction-level parallelism (ILP). Figure 1.2 depicts an overview of superscalar processor and VLIW processor as well as scalar processor. In a scalar processor, an operation (or instruction) is fetched from the instruction memory every cycle and assigned to a functional unit (FU or hardware resource) to execute it. In a superscalar processor, multiple operations are fetched from the instruction memory at a time and then a special hardware unit analyzes a parallelism among the operations. Then operations which can be executed in parallel are issued and executed. On the other hand, a VLIW processor fetches a VLIW instruction which contains multiple operations that can be executed in parallel. Since a compiler has extracted parallelism and scheduled operations, operations in a VLIW instruction can be issued with a simple hardware, while a superscalar processor dynamically analyzes parallelism and schedules operations using a special hardware.

Since a set of operations that can be executed in parallel varies in different VLIW architectures, object code of VLIW processor has no compatibility among different architectures. Binary compatibility of object code is, however, not necessarily important in embedded sys-

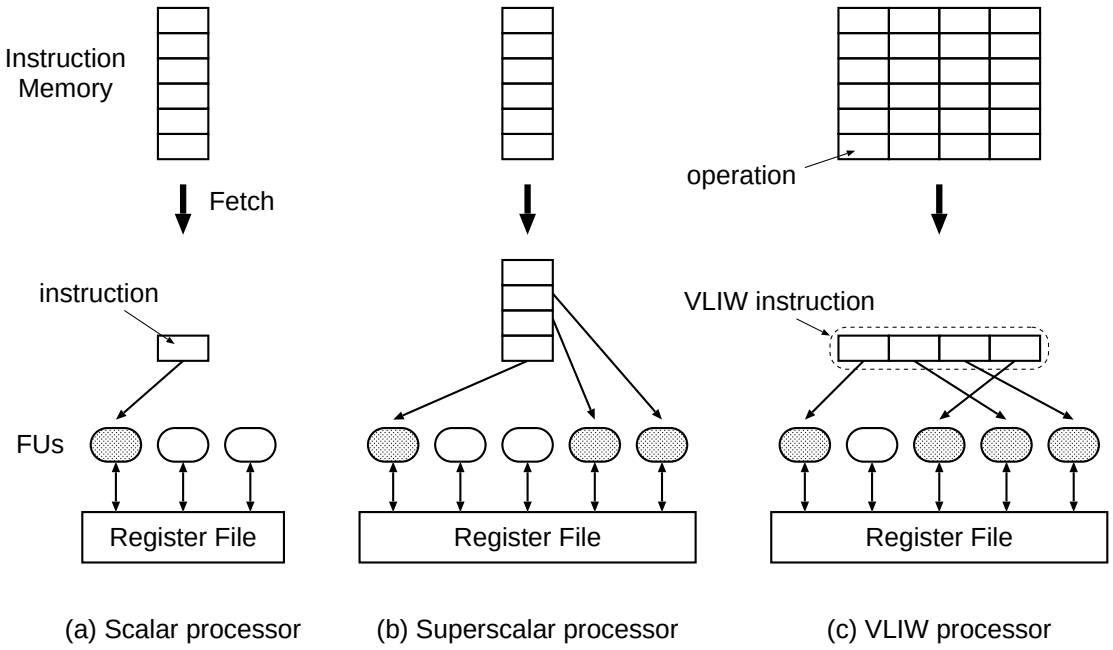


Figure 1.2: Overview of processor architectures.

tems, since software is typically provided together with a target system (hardware) in embedded systems and it can be compiled again in case that the target system is changed, while software and hardware are provided independently in personal computers. And also, a VLIW processor does not need the special hardware for operation scheduling, hence it has simpler hardware than a superscalar processor and it leads to less power consumption. Therefore, a VLIW ASIP is a perfect solution for embedded systems that require low energy consumption as well as high performance.

1.2 Design Challenges

Designing a low energy system using VLIW processor is, however, a challenging problem.

1.2.1 VLIW processor design challenges

Designing a VLIW processor is usually more complex than designing a scalar processor. A complex system makes a design time longer and error-prone. Therefore, a technique for improving the design productivity of VLIW processors is required.

VLIW architecture has many architecture parameters, such as the number of issue slots, the number of functional units. A dispatching rule, which represents which slot issues a certain operation and which combination of operations is allow to be executed at a time, is also an important parameter in VLIW architecture; an unprofitable dispatching rule, where the combination is not so much used, simply makes the hardware logic complex. Since it is difficult to properly define these parameters for the target application in a straightforward way, design

space exploration is commonly used, where designing and evaluating are iterated for a lot of architectures to determine an optimal parameter set. Thus, the technique for improving the design productivity is very important in terms of design space exploration as well.

1.2.2 Low power embedded systems challenges

Low energy is also a significant factor in design of modern VLIW processors. Traditionally, VLIW processors are beset by structural problems; a large size of object code, and complexity of logic compared with scalar processors. Power analysis of VLIW processor reveals that the significant amount of energy is consumed in the instruction memory hierarchy. For example in Lx processor, a VLIW processor designed by Hewlett-Packard and STMicroelectronics, up to 40% of the total processor energy is consumed in the instruction caches alone [8].

An L0 buffer (a.k.a. loop buffer) is an efficient technique to reduce energy consumption in the instruction memory hierarchy [9, 10, 11]. In most embedded applications, significant amount of execution time is spent in small program segments (which consist of loops). An L0 buffer stores these small program segments in a small buffer (SRAM or register file based) instead of a big instruction cache. Then the processor core only accesses to the buffer during the loop execution. This reduces the number of accesses to the higher level of the instruction memory hierarchy and therefore giving large energy reduction, for instance up to 60% as shown in [10].

In spite of such loop buffering techniques, the instruction memory remains a major power bottleneck in most VLIW processors because the conventional centralized loop buffer architecture is not so efficient. Despite adding it, an L0 buffer in VLIW processor consumes significant energy: about 20% in an 8 slot VLIW processor [12].

Since a VLIW processor does not issue operations from all slots in every cycle due to the limitation of ILP (instruction level parallelism) of application, some access to the instruction memory hierarchy is not necessary in some cycles. In order to reduce this unnecessary access, L0 cluster technique is proposed [13, 14].

The approach of L0 cluster can reduce energy consumption of embedded systems using VLIW processor, however, the optimal configuration of L0 cluster is very sensitive to a target application. Hence, an approach to efficiently obtain an optimal configuration is important in a low energy design of VLIW processors.

1.3 Related work

Some methods are proposed so far for low energy and the design productivity of processors.

In these years, designing processors using HDL (hardware description language) is a common way, since a target design has become larger and more complex. The design using HDL (e.g. VHDL, Verilog HDL) in register transfer level (RTL or RT level) is widely accepted and it promises much better design productivity than designing in gate level or transistor level that are used few decades ago. However, the size of design that can be implemented on a single silicon chip increases along with the advance of deep submicron technology. Therefore, it requires a further aggressive approach to design circuitry in higher level.

A processor design method that uses a base processor is one of the promising approaches that enable higher productivity in processor design. This method assumes a certain processor architecture as a base architecture and generates a derivative according to some additional specifications given as input. Since only extension to the base processor is needed to be specified, designers can obtain the target processor design in relatively shorter time. The flexibility of architecture is, however, limited because basic architecture parameters such as pipeline architecture are bounded to the base processor.

Another category of processor generation methods that help the design productivity of processor is known as architecture description language (ADL). The method takes higher level of description than RTL as input, which is usually a smaller amount of description than HDL. Then the method generates an HDL description which represents a structure that a designer intended to design. Since ADL supports a wide variety of architectures that should be evaluated in a phase of design space exploration, an approach using ADL is attractive for processor designers who seek the next generation of design methods.

A lot of researches are done on low energy processor design. One of the well-known techniques for the low energy requirement is clock gating. Since most of energy consumption is due to switching activity of wire or register, by gating a clock supply to a functional unit that is unused, unnecessary switching activity can be suppressed. Dynamic voltage and frequency scaling [15], substrate biasing, and power shut-off are also known as a technique to reduce energy consumption.

Besides the hardware approaches, some researches target software approaches for the low energy problem. One idea is to control voltage and frequency from software. Modern operating systems have such capability which decreases the clock frequency of processor when a processor is idle. Some researches target data locality in the data cache and tries to make data access more efficient [16].

As an approach to reduce power consumption in the instruction memory hierarchy, which consumes significant power in embedded systems, an L0 buffer or loop buffer [9, 10, 11] is a well known architecture. The buffer locates between the instruction cache and the processor core. By storing frequently executed code, that is loops, into a small buffer, it benefits energy reduction as well as performance improvement. Decode filter cache [17] is an architecture to reduce the power consumption of instruction decode as well as instruction fetch. Since decoding instructions also consumes significant power as well as fetching them, the approach tries to store decoded information into the buffer.

Some compiler techniques are also proposed to increase the utilization of loop buffer for further improvement of energy efficiency. A method proposed in [18] transforms code that cannot be executed on a loop buffer as it is. By applying some code transformation technique such as conditional instructions, the utilization of loop buffer can be improved. Another approach [19] optimizes software to efficiently use a loop buffer. The approach optimizes software using if-conversion and increases the utilization of loop buffer, and consequently reduces the instruction fetch power by 72%.

1.4 Approach for low power embedded systems using VLIW processors

This thesis proposes methods for each challenge discussed in Section 1.2. For the first challenge about VLIW processor design productivity, this thesis proposes a synthesizable HDL generation method for configurable VLIW processors, which takes a processor specification description as input and generates a synthesizable HDL description of a target VLIW processor. The proposed approach allows a designer to change the number of slots and pipeline stages, dispatching rules, and so on. Control and decode logic, and the data-path of a target VLIW processor are automatically generated from the processor specification description.

As the second method for low power design, an algorithm of operation shuffling is proposed. The algorithm generates and evaluates a lot of schedules for a target application, and it finds an energy efficient schedule. To reduce the exploration space, some heuristics are also proposed.

In order to minimize the iteration of operation shuffling, this thesis then proposes an efficient scheduling method that generates a low energy schedule for a given cluster configuration.

1.5 Main contributions

This thesis first describes a generation method for configurable VLIW processors in Chapter 3. Though VLIW processors are effective solution for embedded systems which require both of high performance and low energy, there are a lot of architectural parameters to be decided by designers. Since these parameters significantly affect the performance and area, it is required to perform the design space exploration where designers evaluate many architectures to determine the optimal parameter set. However, designing a VLIW processor is very complex, and consequently time consuming and error-prone. Hence, design space exploration on VLIW processors could not have been performed efficiently so far. The VLIW processor generation method described in Chapter 3 supports a flexible architecture model, especially in dispatching rules. Therefore, this method enables the design space exploration on a wide variety of VLIW architectures with high design productivity. Figure 1.3 depicts a system model using a VLIW processor. Chapter 3 focuses on the design productivity for VLIW processors as shown in Fig. 1.3 (a).

An operation shuffling algorithm is then described in Chapter 4. This algorithm improves energy efficiency by changing operation scheduling. Since an L0 cluster configuration is very sensitive to operation scheduling, various schedules should be evaluated in order to obtain an optimal schedule. By shuffling all basic blocks iteratively, energy consumption can be reduced significantly. To reduce the size of the exploration space, some heuristics are also described in Chapter 4.

Since a simple operation shuffling takes huge amount of time even if the above heuristics are applied, a more efficient method that finds a low energy operation schedule is then described in Chapter 5. By exploiting some scheduling algorithms described in the chapter, a compiler can generate a low energy schedule in a straightforward way. Chapter 4 and 5 focus on a low energy methods for the instruction memory hierarchy as shown in Fig. 1.3 (b).

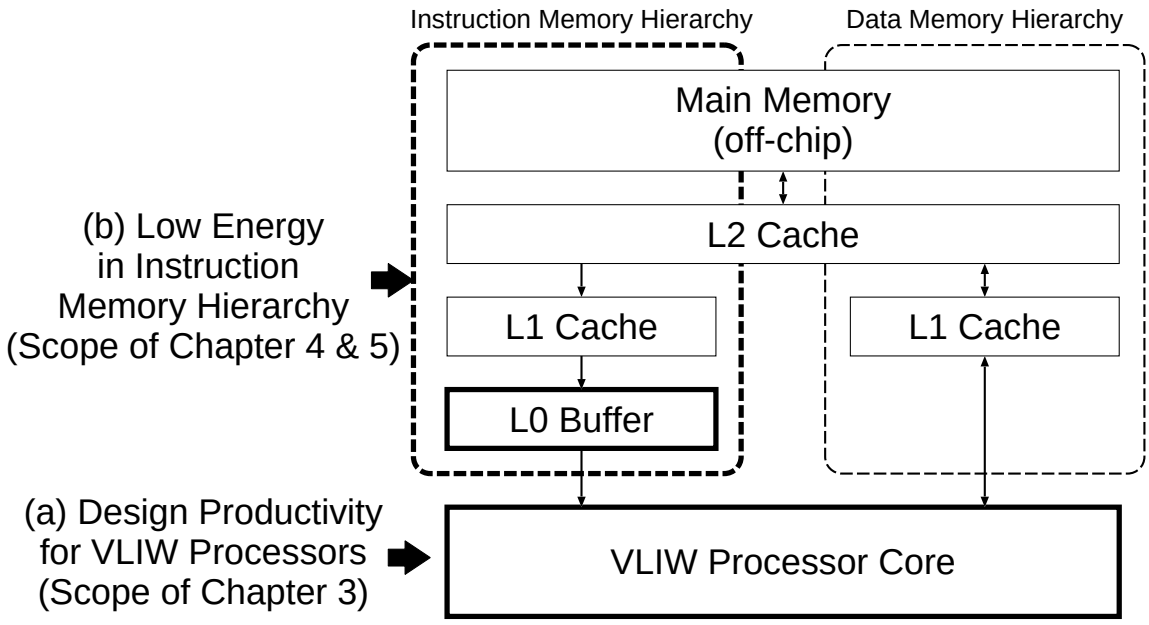


Figure 1.3: System model using a VLIW processor.

1.6 Thesis organization

The rest of this thesis is organized as follows. Chapter 2 discusses related work. Chapter 3 describes a synthesizable HDL generation method for configurable VLIW processors. In Chapter 4 an operation shuffling algorithm for low energy embedded systems using VLIW processor is proposed. Chapter 5 describes an efficient method for low energy operation scheduling. Finally, Chapter 6 concludes this thesis.

Chapter 2

Related work

This chapter describes the previous studies on hardware generation and low energy technique for embedded systems.

2.1 Approaches for hardware generation

An embedded system can be typically implemented by using ASICs (Application Specific Integrated Circuits), GPPs (General Purpose Processors), or ASIPs (Application Specific Instruction-set Processors). ASICs have advantages in the performance per area and power consumption compared with general purpose processors. However, the flexibility of ASICs is very low since the circuits are almost fixed for a specific application. On the other hand, GPPs have the flexibility as they are fully programmable. GPPs are, however, usually not optimal for a target application in terms of performance and power consumption. Hence, GPPs sometimes do not meet a performance requirement and often exceed a power limitation of embedded systems. ASIPs can fulfill both the requirements for flexibility as they are programmable, and for performance as they have an application specific instruction set. Therefore, ASIPs are an appropriate solution for embedded systems.

Configurable processors are often used for designing ASIPs. Configurable processors have some parameters to tune up their instruction set, such as the bit width of data path, the number of general purpose registers, and additional instructions. By using configurable processors, the design time can be shorter than the manual design since the basic architecture is almost fixed. While configurable processors can be more general purpose oriented, it can be more energy efficient or higher performance when their instruction set is tuned. Therefore, configurable processors are reasonable solution for the design of ASIPs.

Various attempts for configurable processors to efficiently design ASIPs and make retargetable compiler and other software tools have been made [20]. Approaches for generation of configurable processors are classified into two categories as follows.

1. Approaches using a base processor (PEAS-I [21], MetaCore [22, 23], Xtensa [24])

2. Approaches using a processor specification description (ISDL [25, 26, 27], nML or Target [28, 29], LISA [30, 31, 32], EXPRESSION [33, 34, 35])

In this section, a brief overview of these approaches is presented.

2.1.1 Approaches using a base processor

PEAS-I [21] uses a basic CPU called PEAS-I CPU. PEAS-I CPU includes an ALU, a shifter, a multiplier, and a divider. Based on the results of application profiling, hardware algorithm of multiplier and divider is selected and unused instructions are omitted automatically. However, designers cannot change architecture of pipeline stage.

MetaCore [22, 23] is an environment to develop an ASIP for digital signal processing. Although MetaCore allows using basic instructions, optional instructions, and user-defined instructions, the number of pipeline stages are fixed. Moreover, MetaCore does not support VLIW architectures.

Xtensa [24] utilizes a customizable RISC processor core, and designers can add a new instruction using special language to improve performance. It is, however, impossible to freely change a pipeline structure.

An approach using a base processor is a method to design an ASIP based on a base processor core by changing the number of registers and adding custom instructions. This approach has an advantage in reduction of design labor, however, has a disadvantage in lack of flexibility to change the instruction bit width or the number of pipeline stages.

2.1.2 Approaches using an architecture description language

ISDL [25, 26, 27] is an instruction set description language for VLIW processors, which can generate software tools. However, the pipeline structure is not so flexible, since an architecture is generated from a highly abstracted description based on the supposed pipeline structure.

nML [28, 29] focuses on the instruction set. The abstraction level of the nML language is in a programmer's model of target processor. It is easy to modify the instruction set, however, the detailed architecture model is hard to specify due to the high abstraction level. And also, a generation method for VLIW architecture is not reported.

EXPRESSION [36, 33] allows designers to describe detailed specification that can represent VLIW processors as well. EXPRESSION can generate simulators and compilers for rapid design space exploration, however, a synthesizable HDL generation method for VLIW processors, especially the support for FU sharing, is not reported.

The approach of LISA [30, 31, 32] can describe architectures considering pipeline structure, and can design VLIW architectures or superscalar architectures. However, a pipeline controller supporting a pipeline stall is not generated unless operations for pipeline registers are explicitly described. Furthermore, though LISA can generate synthesizable HDL description for control logic, it is not reported to generate entire data-path of processor.

Although an approach using an ADL description has disadvantages in increase of description, it can describe a detailed processor specification and generate various architectures.

In design of ASIPs, a method that allows various pipeline structures and that can generate a control logic, that is usually error-prone in manual design, is desirable. Unfortunately, there has not been any method that can generate VLIW processors effectively and flexibly. Therefore, this thesis proposes a VLIW processor generation method from a processor specification description to specify a detail of processor and to explore large design space of VLIW processors.

2.2 Overview of low power optimizations for embedded processors

There are a lot of researches on low energy processor design. One of the well-known techniques for the low energy requirement is clock gating. Since most of energy is consumed due to switching activity of wire or register, by gating a clock supply to a functional unit that is unused, unnecessary switching activity can be suppressed. Dynamic voltage and frequency scaling, substrate biasing, and power shut-off are also known as a technique to reduce energy consumption.

Besides the hardware approaches, some researches target software approaches for the low energy problem. One idea is to control voltage and frequency from software. Modern operating systems have such capability which decreases the clock frequency of processor when a processor is idle. Some researches target data locality or layout in the data cache and tries to make data access more efficient [16]. A compiler-guided low power method for scratch pad memories [37] optimizes memory-data layout to maximize bank idleness of scratch pad memories.

As an approach to reduce power consumption in the instruction memory hierarchy, which consumes significant power in a processor, an L0 buffer or loop buffer [9, 10, 11] is well known. The buffer locates between the instruction cache and the processor core. By storing frequently executed code, e.g. inner most loops, into a small buffer, it benefits energy reduction as well as performance improvement. Decode filter cache [17] is an architecture to reduce the power consumption of instruction decode as well as instruction fetch. Since decoding instructions also consumes significant power as well as fetching them, the approach tries to store decoded information into the buffer.

For further improvement of energy efficiency in loop buffers, some compiler techniques are also proposed, which try to increase the utilization of loop buffer. A method proposed in [18] transforms code that cannot be executed on a loop buffer as it is. A loop that has a transfer of control (e.g. branch operation) inside or a loop where the number of iterations is unknown cannot be executed on a loop buffer. By applying some code transformation technique such as a conditional instruction or an explicit manipulation of the loop counter, the utilization of loop buffer can be improved. Another approach [19] optimizes software to efficiently use a loop buffer. The compiler technique optimizes software using if-conversion and increases the utilization of loop buffer, and consequently reduces the power of instruction fetch by 72%.

2.3 Low power optimization on instruction memory hierarchy

An L0 buffer (a.k.a. loop buffer) is an efficient technique to reduce energy consumption in the instruction memory hierarchy [9, 10]. In most embedded applications, significant amount of execution time is spent in small program segments (which consist of loops). The technique stores these small program segments in a small L0 buffer (SRAM or register file based) instead of the big instruction cache. Then the processor core only accesses to the buffer during the loop execution. This reduces the number of accesses to the higher level of the instruction memory

hierarchy and therefore giving large energy reduction, for instance up to 60% as shown in [10].

In a simple application of the monolithic L0 buffer to a VLIW processor, at each cycle the operations would be fetched for all the slots of the VLIW from the monolithic L0 buffer. However, such a monolithic L0 buffer is not effective as not all slots are always active. This implies that some slots would require unnecessary buffer access for NOP operation [13]. Hence, L0 cluster generation was proposed to obtain a low energy system [13, 14].

In a clustered VLIW processor, 'clustered' usually refers to data-path clusters. For example, the TI C6X processor has clustered data-path [38], which can issue up to eight operations at a time, and has two separated register files. If many FUs are connected to a monolithic register file, it leads to significant increase of delay time, area, and power consumption. Therefore, many researchers have tried to cluster FUs and divide a register file in order to decrease the number of FUs connected to each register file. The clustered register file reduces the number of ports of register file, which reduces the delay, area, and energy of the register file. Clustered register file is almost always used in VLIWs with a larger number of issue slots [39, 40]. Note that data clustering may cause increase of execution cycles since sometimes a copy operation is needed to move data between register files, while instruction clustering does not cause such a problem.

Some VLIW architectures like Lx processor have a notion of instruction clustered instruction fetch [41]. However, the instruction clusters correspond directly to the data clusters, while this thesis makes an instruction cluster explicitly separated from a data cluster to increase the exploration freedom; an L0 cluster can be applied independently for data clusters. This results in larger gains of up to energy reduction of 75% as demonstrated in [14].

A loop buffer or loop cache has been studied for years, which aims for energy reduction on instruction memory hierarchy as well as performance improvement. By exploiting such a loop buffering mechanism, L1 or higher cache access rate is reduced by up to 38% [9], and it leads energy reduction of overall instruction memory of up to 67% [10]. A loop buffer is implemented as a register file or SRAM based architecture and access to buffer is fully controlled by a control unit (e.g. LC and ITC, described in Section 4.2), while a loop cache has possibility of cache miss whose performance penalty cannot be accepted in certain embedded applications.

Operation shuffling for instruction clusters has been studied only in the recent past [42, 43]. Similar to the work presented in this article, their objective is an overall hardware/software energy reduction for embedded VLIW processors. However, their target is on the instruction cache, while this thesis focuses on L0 buffers and generation of L0 clusters.

Chapter 3

Hardware generation for VLIW processors

This chapter describes the proposed VLIW processor generation method. First, a target VLIW processor model is explained. Secondly, a scalar processor generation method that the proposed VLIW processor generation method is based on is introduced. Finally, the proposed VLIW processor generation method is described.

3.1 Problem and motivation

Designing a VLIW processor is usually more complex than a scalar processor which issues only one operation at a time. A complex system makes a design time longer; describing complex control logic of such a system is tedious and error-prone. Therefore, a technique for improving the design productivity of VLIW processors is required.

VLIW architecture has many architecture parameters, such as the number of issue slots, the number of functional units. A dispatching rule, which represents which slot issues a certain operation and which combination of operations is allowed to be executed at a time, is also an important parameter in VLIW architecture; an unprofitable dispatching rule, where the combination is not so much used, simply makes the hardware logic complex. Since it is difficult to define these parameters appropriately for the target application in a straightforward way, design space exploration is commonly used, where designing and evaluating a lot of architectures to determine an optimal parameter set. Thus, the technique for improving the design productivity is very important in terms of design space exploration as well.

3.2 VLIW processor model

This section introduces a VLIW processor model [44] that the proposed VLIW processor generation method uses. This model can represent various architecture of VLIW processors.

A VLIW instruction consists of multiple *operations* that are executed simultaneously. *Dispatching* is a managing process to assign issued operations to appropriate FUs A *Slot* is a unit to issue an operation. A VLIW processor has one or more slots and issues multiple operations

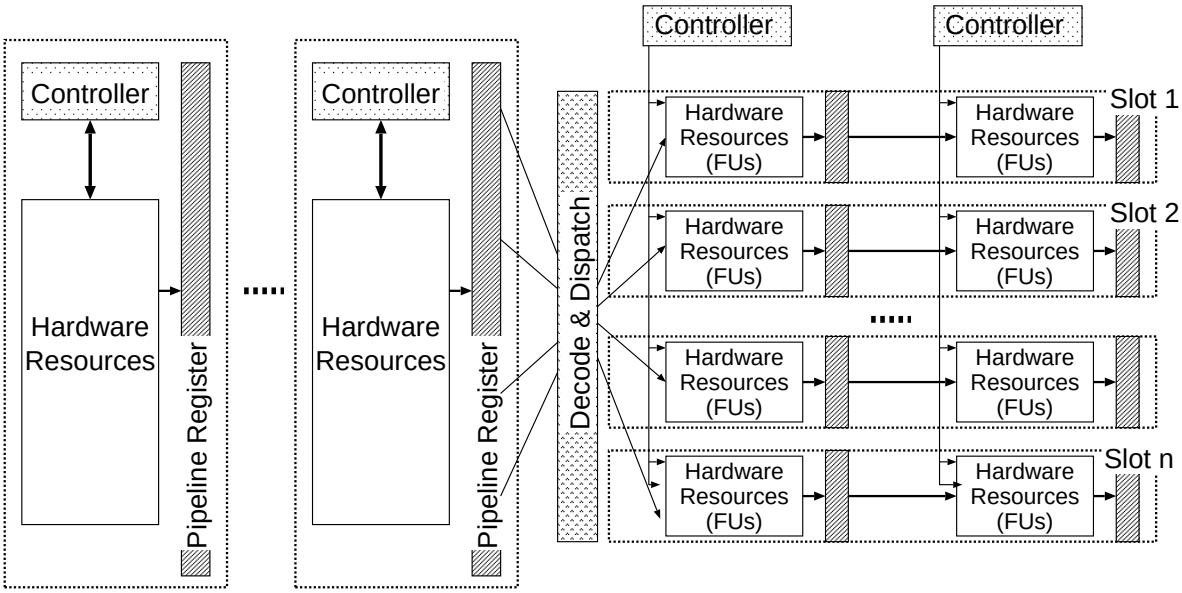


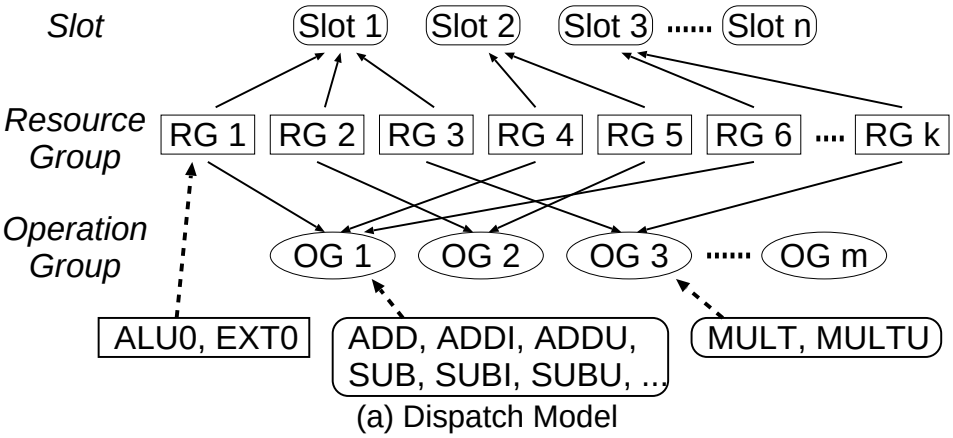
Figure 3.1: VLIW processor model.

from the slots in parallel. Note that a VLIW processor with one slot is equivalent to a scalar processor in this model.

Figure 3.1 illustrates a hardware model of VLIW processor. In first few pipeline stages, a VLIW instruction is fetched, and then it is decoded and operations are dispatched into FUs. A data-path in each pipeline stage consists of *hardware resources*, that are mainly combinational circuits, and pipeline registers to send data into the next pipeline stage. An FU is a kind of a hardware resource. In normal operation mode without any pipeline interlock, hardware resources receive input data from the previous pipeline registers, and send output data into the next pipeline registers.

3.2.1 Dispatching model

Figure 3.2 (a) shows the dispatching model of [44]. To represent complex dispatching rules in a simple description, two concepts, *operation group* and *resource group*, are introduced. An *operation group* is a set of operations that have the same characteristic on dispatching, for instance, a member of operation group can be executed on the same kind of FUs. A *resource group* is a set of FUs that are used when a certain operation is executed in a certain slot. Note that a resource group belongs to one slot and one operation group. In Fig. 3.2 (a), operations ADD, ADDI, and so on are members of operation group OG1. Resource group RG1 consists of FUs ALU0 and EXT0. RG1 belongs to OG1 and Slot1. An FU can belong to one or more resource groups, which means that a shared FU is represented by belonging to multiple resource groups. In this way, dispatching rules are described using three relations; between slots and operation groups, between slots and resource groups, and between operation groups and resource groups. Figure 3.2 (b) shows a dispatching rule description of the above model.



(a) Dispatch Model

```
slot_opegroup {
  { Slot1: OG1, Slot2: OG1, Slot3: OG3, ... },
  { Slot1: OG1, Slot2: OG2, Slot3: OG1, ... },
  ...
};
opegroup_resgroup {
  OG1: RG1, RG4, RG6;
  OG2: RG2, RG5;
  ...
};
slot_resgroup {
  Slot1: RG1, RG2, RG3;
  Slot2: RG4, RG5;
  ...
};
```

(b) Dispatch Rule Description

Figure 3.2: Example of the dispatching model.

Micro-operation description to be explained later is described for a pair of a resource group and an operation. This method allows to represent a wide range of dispatching rules as a designer intends to make.

3.2.2 Interrupt model

The proposed interrupt model of VLIW processor is shown below.

- Supported interrupt type and instruction canceling policy (in descending order of priority)
  - Reset interrupt (the highest priority): *cancel all instructions policy*
  - Nonmaskable interrupt: *cancel all instructions policy*

  - Internal interrupt: *cancel descending instructions policy*
  - External interrupt (the lowest priority): *wait all instruction completion policy*
- Precise interrupt
- Select the highest-priority interrupt among multiple interrupts occurred at a time

The proposed interrupt model is based on the interrupt model of a scalar processor [45]. *Cancel all instructions policy* annuls all operating instructions and starts interrupt processing immediately after an interrupt request signal is asserted. *Cancel descending instructions policy* annuls only descending instructions when an interrupt occurs, and starts interrupt processing after completion of ascending instructions. *Wait all instruction completion policy* annuls no instructions but suppresses fetching a new instruction, and interrupt processing is started after completion of all instructions that were being executed when an interrupt request signal was asserted.

3.3 Hardware architecture of targeted VLIW processor

This section describes an overview of target VLIW processor and execution model of the processor.

3.3.1 Hardware overview of VLIW processor

Figure 3.3 illustrates control paths of scalar processor and VLIW processor. As shown in Fig. 3.3 (a), a data-path of scalar processor is controlled by decode signals that are generated from the value of the instruction register. On the other hand, in a VLIW processor, $Dec_{rg,ope}$, *decode signal for resource group and operation*, is generated from the value of the instruction register, $Dec_{slot,opeg}$, *decode signal for slot and operation group*, is generated from combinations of $Dec_{rg,ope}$, Det_{ptrn}, *detection signal for VLIW pattern*, is generated from combinations of $Dec_{slot,opeg}$, $Actv_{rg}$, *resource group activation signal*, is generated from combinations of Det_{ptrn}. Then, a VLIW processor controls a data-path by $Actv_{rg}$ and $Dec_{rg,ope}$, as shown in Fig. 3.3 (b).

A decode signal for resource group and operation ($Dec_{rg,ope}$) is a signal to identify an operation in the instruction register. As was mentioned earlier, since micro-operation description is described for a pair of a resource group and an operation, this signal and a micro-operation description have one-to-one mapping. For instance, if operation ADD executing on resource group $RG1$ exists, a signal corresponding to its micro-operation is $Dec_{RG1,ADD}$. A decode signal for slot and operation group ($Dec_{slot,opeg}$) is a signal to identify an operation group to be issued from the slot. $Dec_{Slot1,ALU}$ is a signal representing that an operation in operation group ALU is issued from $Slot1$. A detection signal for VLIW pattern (Det_{ptrn}) is a signal to identify a VLIW pattern in the instruction register, for instance, a signal representing the third pattern is Det_3. A resource group activation signal ($Actv_{rg}$) is a signal representing that the resource group is assigned to a detected VLIW pattern, for instance, an activation signal of $RG1$ is $Actv_{RG1}$. In case that multiple candidates of resource group exist for an operation

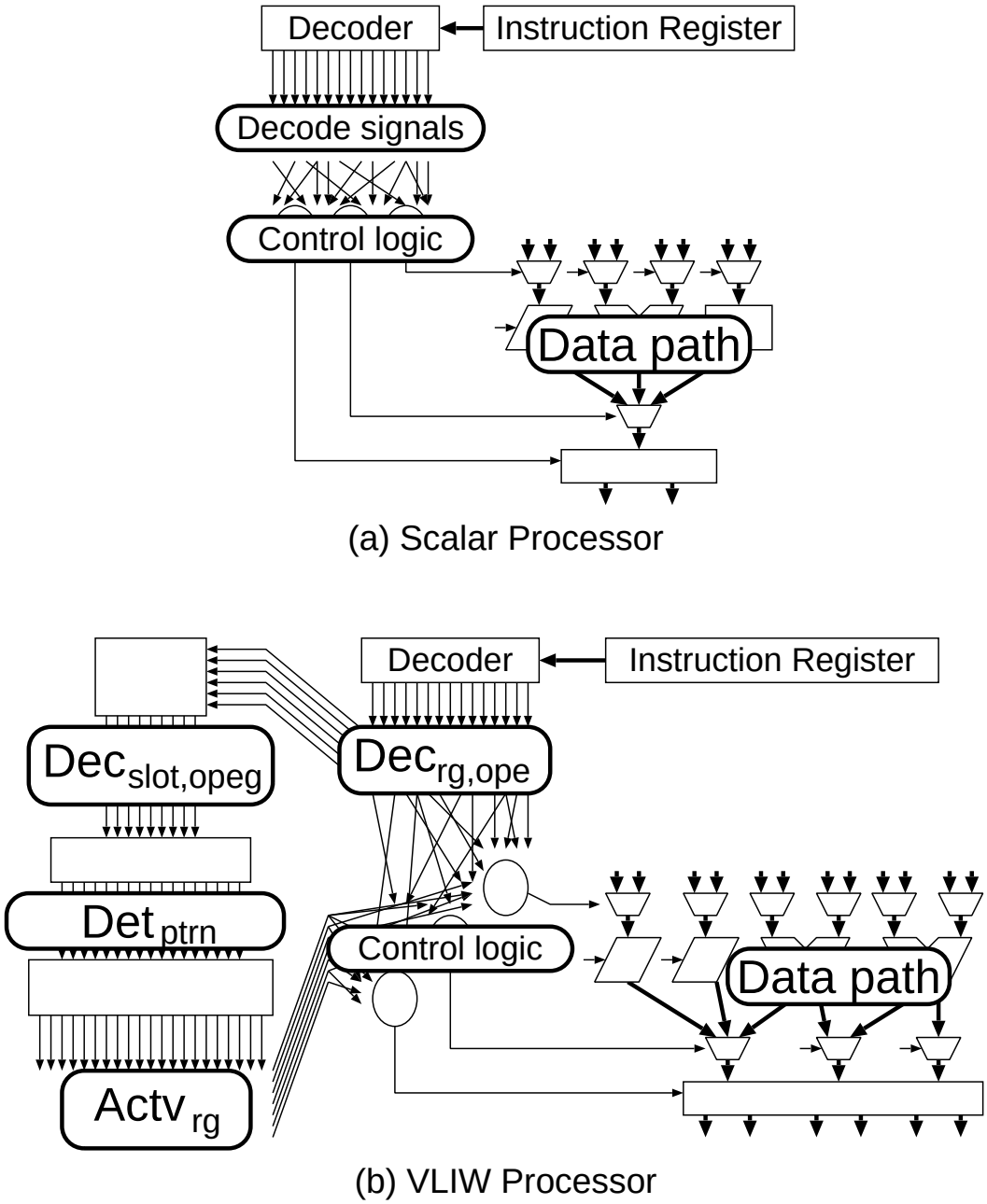


Figure 3.3: Control paths of scalar processor and VLIW processor.

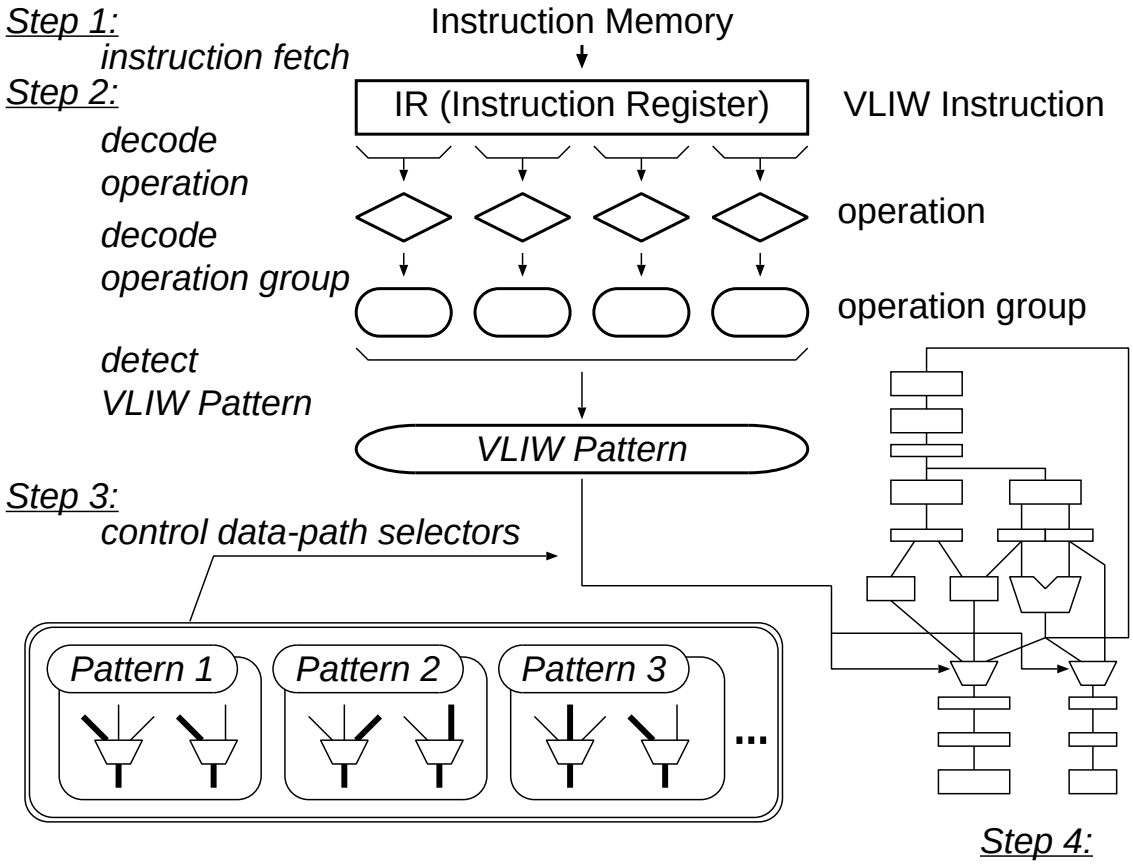


Figure 3.4: Execution model of VLIW processor.

in a certain slot, all corresponding signals $Dec_{rg,ope}$ become active, however, only one signal $Actv_{rg}$ is active. Then an operation can be successfully executed without any conflict of FU.

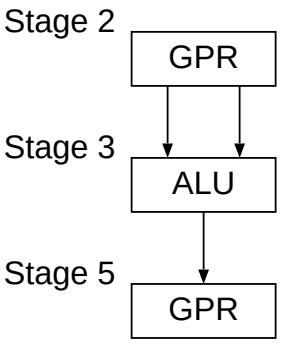
3.3.2 VLIW Processor Execution Model

Figure 3.4 illustrates the execution model of VLIW processor. In the proposed model, a VLIW processor runs with repetition of steps as follows. First, a VLIW instruction is fetched from an instruction memory, and stored to the instruction register. Secondly, operations in the VLIW instruction are decoded. Operation groups for each operation are obtained, and a VLIW pattern is detected according to a combination of the operation groups. Thirdly, a data-path is controlled by switching data-path selectors according to prepared information corresponding to the detected VLIW pattern. Finally, operations are executed in the data-path controlled in the previous step.

In the proposed method, FUs assigned to a VLIW pattern are determined first, and then control signals of a data-path are generated using the information of assignment.

```
micro_operation ADD {
 wire [31:0] src0;
 wire [31:0] src1;
 wire [31:0] res;
 stage 2 {
  src0 = GPR.read0(rs0);
  src1 = GPR.read1(rs1);
 };
 stage 3 {
  wire [3:0] flag;
  <res,flag> = ALU.add(src0,src1);
 };
 stage 5 {
  null = GPR.write0(rd,res);
 };
};
```

(a)



(b)

Figure 3.5: An example of micro operation description and DFG generated from the description.

3.4 Synthesizable HDL generation method for scalar processors

This section introduces a scalar processor generation method [46]. The proposed VLIW processor generation method is based on this method.

In the method of [46], a data flow graph (DFG) corresponding to an instruction is derived from a micro-operation description representing behavior in each pipeline stage of instruction, and then DFGs corresponding to all instructions are merged into one data path that represents an entire processor. Figure 3.5 (a) shows an example of micro-operation description. In Fig. 3.5 (a), behavior in pipeline stage 2, 3, and 5 are described. First, keyword *wire* declares three 32 bit variables, *src0, src1*, and *res*. In stage 2, values of operands, *src0* and *src1*, are read from general purpose register *GPR*. In stage 3, *src0* and *src0* are added by *ALU*, and the result is stored into *res*. In stage 5, the result is written back into *GPR*.

Information of connections between FUs is extracted from the micro-operation description in Fig. 3.5 (a). A DFG in Fig. 3.5 (b) is corresponding to a micro-operation description in Fig. 3.5 (a).

Figure 3.6 shows an example of merging DFGs. Figure 3.6 (a) and (b) are DFGs of an addition operation and a shift operation, respectively. These DFGs are merged into a DFG shown in 3.6 (c). Moreover, since the proposed method is based on [47], the method can control pipeline hazards such as structural hazards.

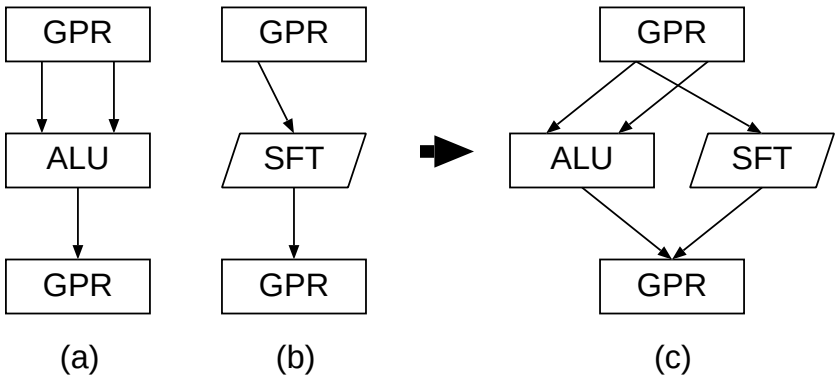


Figure 3.6: An example of merging DFGs.

3.5 Synthesizable HDL generation method for VLIW processors

This section describes the details of the proposed VLIW processor generation method. First, input of the algorithm is explained. Then a generation method of VLIW processor based on the model shown in section 3.2 is proposed.

Let a *VLIW pattern* denote a categorized VLIW instruction that has the same property in dispatching. In the proposed method, FUs dispatched to a VLIW pattern are decided before HDL generation.

3.5.1 Input of VLIW processor generation method

This section defines input of the proposed generation method. First, dispatching rules are defined, secondly, an entire processor specification description is defined.

Let $Slot$ be a set of slots, RG be a set of resource groups, OG be a set of operation groups. $OpegResg$, a relation between an operation group and resource groups, and $SlotResg$, a relation between a slot and resource groups, are represented as follows:

$$og \in OG, OpegResg(og) \subseteq RG, OpegResg(og) \neq \emptyset, \tag{3.1}$$

$$s \in Slot, SlotResg(s) \subseteq RG, SlotResg(s) \neq \emptyset, \tag{3.2}$$

A VLIW pattern, $VLIW\_ptrn$, is represented as follows:

$$s \in Slot, VLIW\_ptrn(s) \in OG. \tag{3.3}$$

$SlotOpeg$ means a set of VLIW pattern as follows:

$$SlotOpeg = VLIW\_ptrn. \tag{3.4}$$

Therefore, a dispatching rule $DispatchRule$ is represented as follows:

$$DispatchRule \quad = \quad \{SlotOpeg, OpegResg, SlotResg\} \tag{3.5}$$

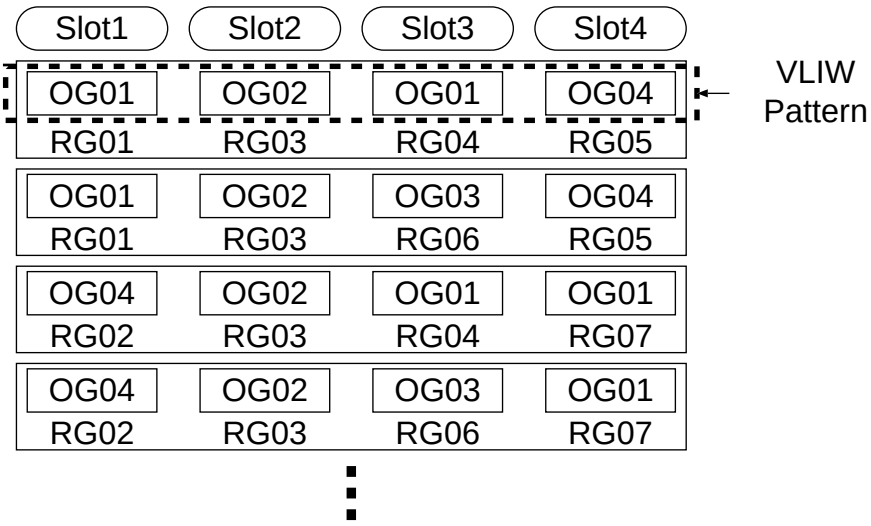


Figure 3.7: Example of table of instruction dispatch pattern T_{IDP}.

Let Res be a set of hardware resources, $Operation$ be a set of definitions of operation that include opecode and operand for each operation, IO be a set of input/output ports of processor, Mod be a set of micro-operation descriptions. Then $Spec$, the input of the proposed VLIW processor generation method, is represented as follows:

$$\begin{aligned} Spec &= \{Slot, Res, RG, Operation, \\ &\quad OG, IO, DispatchRule, Mod\} \end{aligned} \tag{3.6}$$

3.5.2 Instruction dispatch pattern

In the proposed method, assignment of resource groups to a VLIW pattern is determined before HDL generation. We call this assignment an *instruction dispatch pattern*. All of instruction dispatch patterns are gathered into a table of instruction dispatch pattern, T_{IDP}. Figure 3.7 shows an example of T_{IDP}. The first entry of the T_{IDP} represents that resource groups RG01, RG03, RG04, RG05 are used for VLIW pattern {OG01, OG02, OG01, OG04}.

In this section, an assignment method of resource group to VLIW patterns is described. The method consists of two steps as follows.

1. Enumeration of resource groups that can execute an operation in an operation group assigned to a slot.

2. Determination of resource group assignment to VLIW pattern.

The input of this algorithm is the following items; relations between slots and operation groups that are equivalent to VLIW patterns, relations between slots and resource groups, and relations between operation groups and resource groups. The output is T_{IDP}.

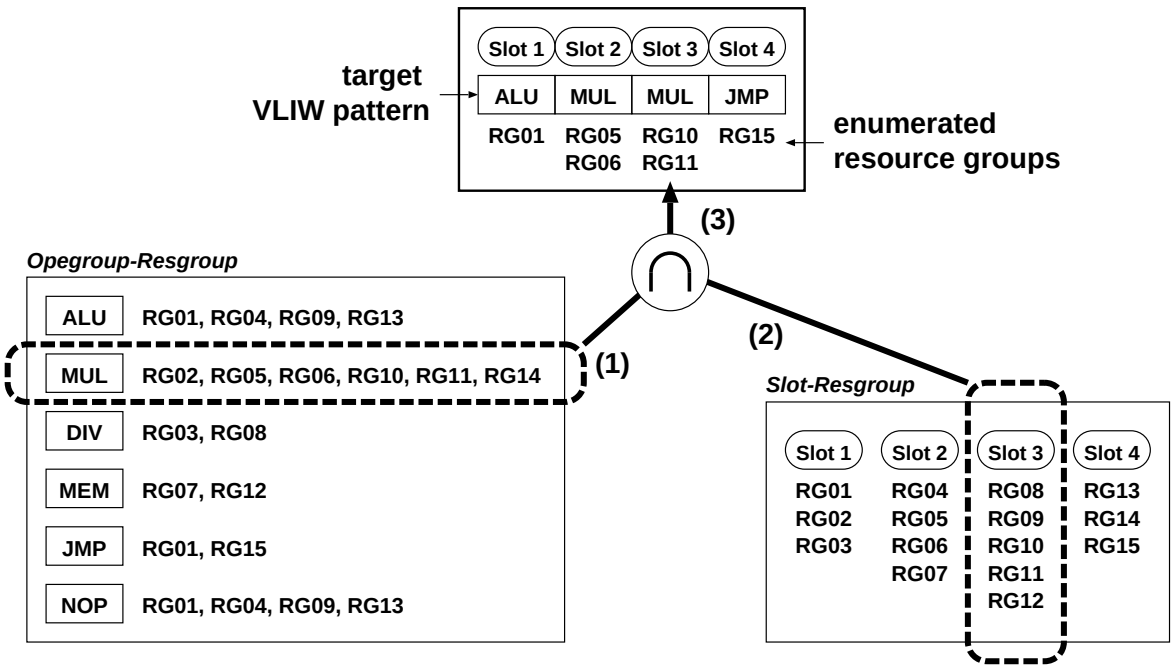


Figure 3.8: Enumeration of resource group for each slot.

(1) Enumeration of resource groups In this step, resource groups, that can execute an operation group in each slot of VLIW pattern, are enumerated.

First, according to relations between operation groups and resource groups, resource groups that can perform the operation groups in each slot of VLIW pattern are calculated. Figure 3.8 shows an example that tries to enumerate resource groups for slot 3 of target VLIW instruction. In Fig. 3.8 (1), a set of resource groups that can execute MUL operation group is fetched.

Then, according to relations between slots and resource groups, resource groups that really belongs to the slot are selected from them. In Fig. 3.8 (2), a set of resource groups that belong to slot 3 fetched, and then intersection of these two sets are calculated in order to determine resource groups available for this VLIW instruction.

(2) Determination of Resource Group Assignment In the resource groups calculated in the previous step, multiple resource groups are sometimes enumerated for a slot. Therefore, it is necessary to determine one resource group for a slot without any conflict of FU among resource groups determined for other slots. In an example of Fig. 3.9, RG05 and RG10 both use the same FU, MUL0. Then, in case that Slot2 uses RG05 or RG06, Slot3 has to use RG11 or RG10, respectively.

Algorithm 1 shows an algorithm that determines one resource group for a slot. FU and Rg are a set of FUs and resource groups, respectively, that are empty at the beginning of this algorithm. RG_s is the resource groups belonging to slot s calculated in the previous step. If $FUs(rg)$, FUs included in rg, do not overlap with FU, $FUs(rg)$ are added to FU and rg is added to Rg. If a combination of resource groups that have no FU conflict is found,

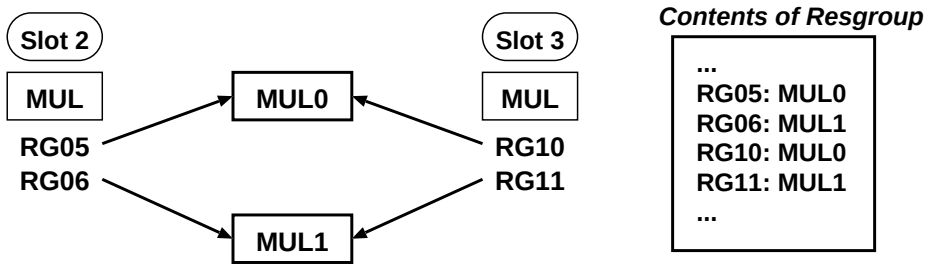


Figure 3.9: Example of FU conflict.

Algorithm 1 Resource group Decision Algorithm.

```
boolean function SelectResgroup(s, FU, Rg) {
  foreach rg in RG_s {
    if ( FUs(rg) ∉ FU ) // no conflict. {
      if ( s.next = null ) {
        Rg = Rg ∪ rg;
        adopt( Rg ); // done.
        return true;
      } else if ( true = SelectResgroup(s.next, FU ∪ FUs(rg), Rg ∪ rg) ) {
        return true;
      }
    }
  }
  return false;
}
```

this algorithm outputs Rg and finishes. The computational complexity of this algorithm is $O(nf^2g^s)$, where s, f, g, n are the number of slots, FUs, resource groups, and VLIW patterns, respectively.

3.5.3 Control signals for dispatching

This section describes a generation method of control signals for dispatching in a VLIW processor.

3.5.3.1 Decode signals for resource group and operation

A decode signal for a pair of a resource group and an operation, $Dec_{rg,ope}$, comprises a logical product of comparisons of opecode and corresponding field in the instruction register.

$$Dec_{rg,ope} = \bigwedge_{opecode \in Opecode_{ope,Slot(rg)}} (IR[Begin(opecode)..End(opecode)] = Value(opecode)), \tag{3.7}$$

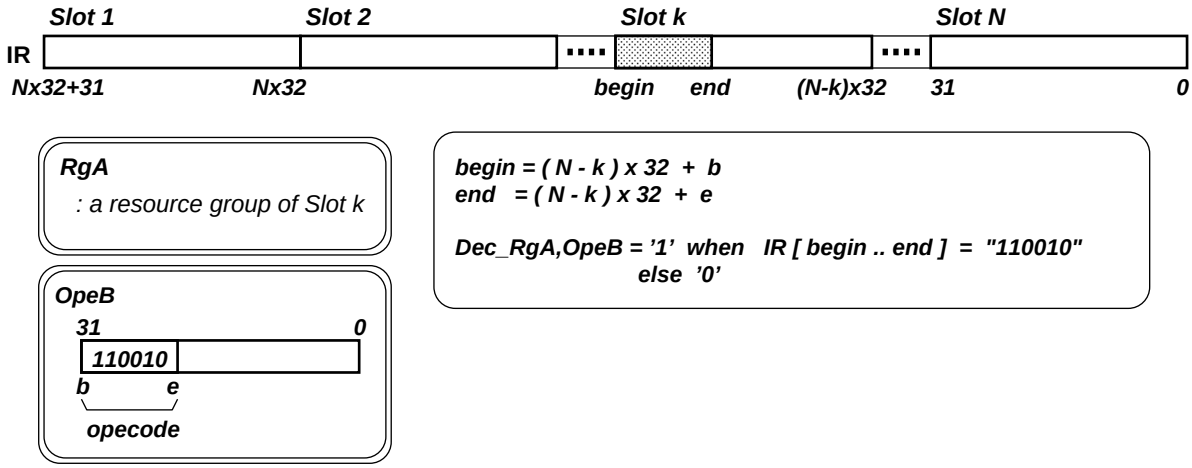


Figure 3.10: Example of decode signal for resource group and operation.

where $Slot(rg)$ returns slot corresponding to rg. $Opecode_{ope,slot}$ includes a set of opecode for ope in $slot$; an opecode consists of value and range of bit field in the instruction register. Figure 3.10 shows an example of $Dec_{rg,ope}$ where the length of operation is 32 bit. In this example, by checking a bit field from $begin$ to end in the instruction register (IR), we can know whether operation $OpeB$ that can be executed on RgA is coming to the instruction register. Note that the decode signal is active if and only if the instruction register contains an operation ope that can be executed on rg.

3.5.3.2 Decode signals for slot and operation group

Let RG_{slot} be a set of resource groups that belong to $slot$. Then $Dec_{slot,og}$, a decode signal for a pair of a slot and an operation group, is represented as a logical sum of $Dec_{rg,ope}$ as follows:

$$Dec_{slot,og} = \bigvee_{\substack{ope \in og \\ rg \in RG_{slot}}} Dec_{rg,ope} \wedge Exist(rg, ope), \tag{3.8}$$

where $Exist(rg, ope)$ is the function that returns true if operation ope executing on rg is defined in input, otherwise returns false.

3.5.3.3 VLIW pattern detection signals

Let $Slot$ be a set of all slots, and $Opegroup(ptrn_{slot})$ be an operation group that corresponds to $slot$ in VLIW pattern $ptrn$.

A detection logic of VLIW pattern $ptrn$, Det_{ptrn}, is represented as a logical product of $Dec_{slot,opeg}$ as follows:

$$Det_{ptrn} = \bigwedge_{\substack{slot \in Slot \\ opeg = Opegroup(ptrn_{slot})}} Dec_{slot,\, opeg}. \tag{3.9}$$

3.5.3.4 Resource group activation signals

Let $Slot(rg)$ be a slot that resource group rg belongs to, Det_{ptrn} be a detection signal of VLIW pattern in instruction dispatch pattern $ptrn$, and $Resgroup(ptrn_{slot})$ be a resource group that corresponds to $slot$ in $ptrn$.

$Actv_{rg}$, an activation signal of rg, is represented as a logical sum of Det_{ptrn} as follows:

$$Actv_{rg} = \bigvee_{ptrn \in T_{IDP}} Det_{ptrn} \wedge (rg = Resgroup(ptrn_{Slot(rg)})). \tag{3.10}$$

3.5.3.5 Control signals of data-path selectors

In the proposed method, a micro-operation description is specified for each pair of an operation and a resource group. Then a data-path of an entire processor is generated by merging DFGs derived from micro-operation descriptions. Since merging often causes signal conflicts at input port of FU, data-path selectors are inserted, so that the conflicts are resolved.

This section describes control logic of the data-path selector, which represents a condition of selection.

A DFG that is derived from a micro-operation description of operation ope executed on resource group rg is valid when the instruction register holds a value that represents ope and rg is activated, as described in Section 3.3. Hence, the logic can be represented by a logical product of an activation signal for rg and an decode signal for a pair of rg and ope. Assume that a DFG that is derived from a micro-operation description of ope executed on rg is $DFG_{ope,rg}$. Then, logic for a selector to form $DFG_{ope,rg}$ is represented as follows:

$$Cond_{DFG_{ope,rg}} = Dec_{rg,ope} \wedge Actv_{rg} \tag{3.11}$$

Figure 3.11 shows merging DFGs and inserting a data-path selector. When operation OpeA executed on resource group ResgM is decoded (identified) and ResgM is activated, the DFG that represents operation OpeA executed on ResgM becomes valid. In other words, the inserted selector is controlled to select the edge derived from the DFG.

By using the activation signal, it is possible to form an appropriate DFG in case that multiple candidates of FU exist for an operation to be executed on the same slot. For example of Fig. 3.11, a certain slot can issue OpeA, however, depending on a combination of operations in other slots, the slot uses either r1 or r2. It is hard to decide only with decode signals which FU is to be used for OpeA; if another slot uses r1, this slot has to use r2, and vice versa.

3.5.4 Control signals for interrupt

Since a reset and an external interrupt are independent from a concept of VLIW processor, such as slot, the same model of [45] can be applied to a VLIW processor. Therefore, this section discusses a model for a nonmaskable interrupt and an internal interrupt.

3.5.4.1 Nonmaskable interrupt

A nonmaskable interrupt (NMI) is the second highest priority interrupt next to the reset interrupt. It is used for an urgent interrupt from the outside of processor, such as a notice of system

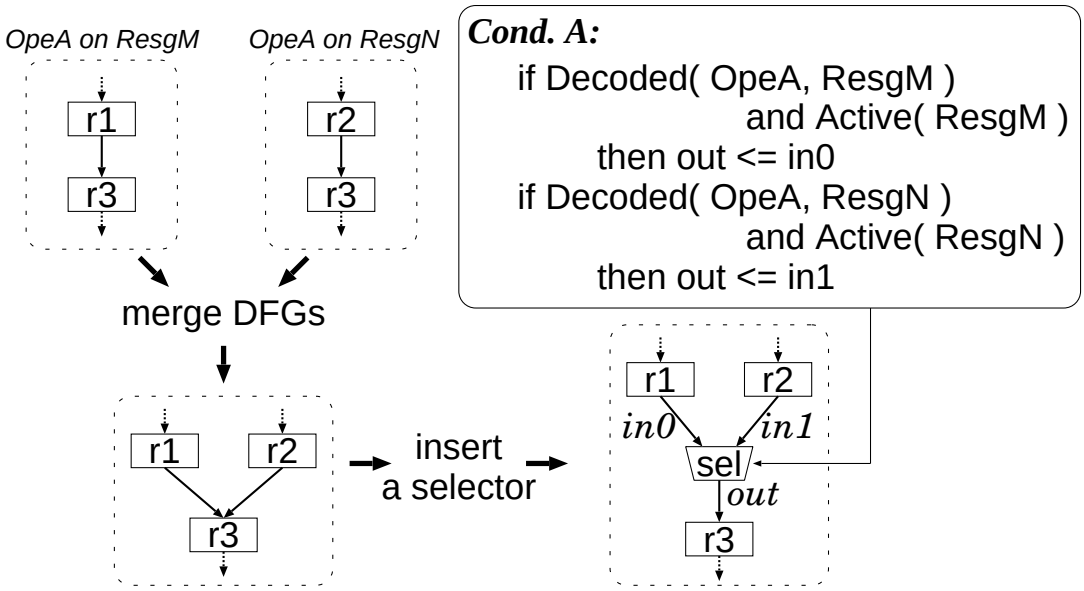


Figure 3.11: Merge of DFGs and selector insertion in the VLIW processor generation.

power down, and so on. Figure 3.12 illustrates a hardware model of interrupt pipeline that can also handle an NMI. In Fig. 3.12, a shaded box represents a pipeline register. *NMI Request* represents a request signal of NMI.

Internal Interrupt Request i, and *Internal Interrupt Code* i represent an internal interrupt request and its identifier in the i-th stage, respectively. In the model of Fig. 3.12, a status of interrupt request and interrupt identifier are entered into the interrupt pipeline. In the model of Fig. 3.12, an external interrupt request signal and its identifier *EXTINT* are entered into the beginning of interrupt pipeline. Since an interrupt request from ascending stages hides an external interrupt request, an external interrupt has the lowest priority. On the other hand, an NMI request signal and a reset interrupt request signal are entered into the end of the interrupt pipeline, then the interrupts have higher priority than other external and internal interrupts. Since a reset interrupt identifier is selected after selecting an NMI, a reset interrupt has higher priority than an NMI.

3.5.4.2 Internal interrupt

In VLIW processors, multiple interrupts can be occurred from multiple slots in a pipeline stage at a time. Therefore, we need a mechanism to select one interrupt to process among multiple interrupts.

Internal Interrupt Model of Scalar Processor In [45], an internal interrupt model of scalar processor is proposed. The detection logic of internal interrupt in the model is shown below. Let Ope_{intr} be a set of possible operations in which internal interrupt $intr$ may occur, $valid_{stage}$ be a condition that represents existence of a valid instruction in $stage$. Then,

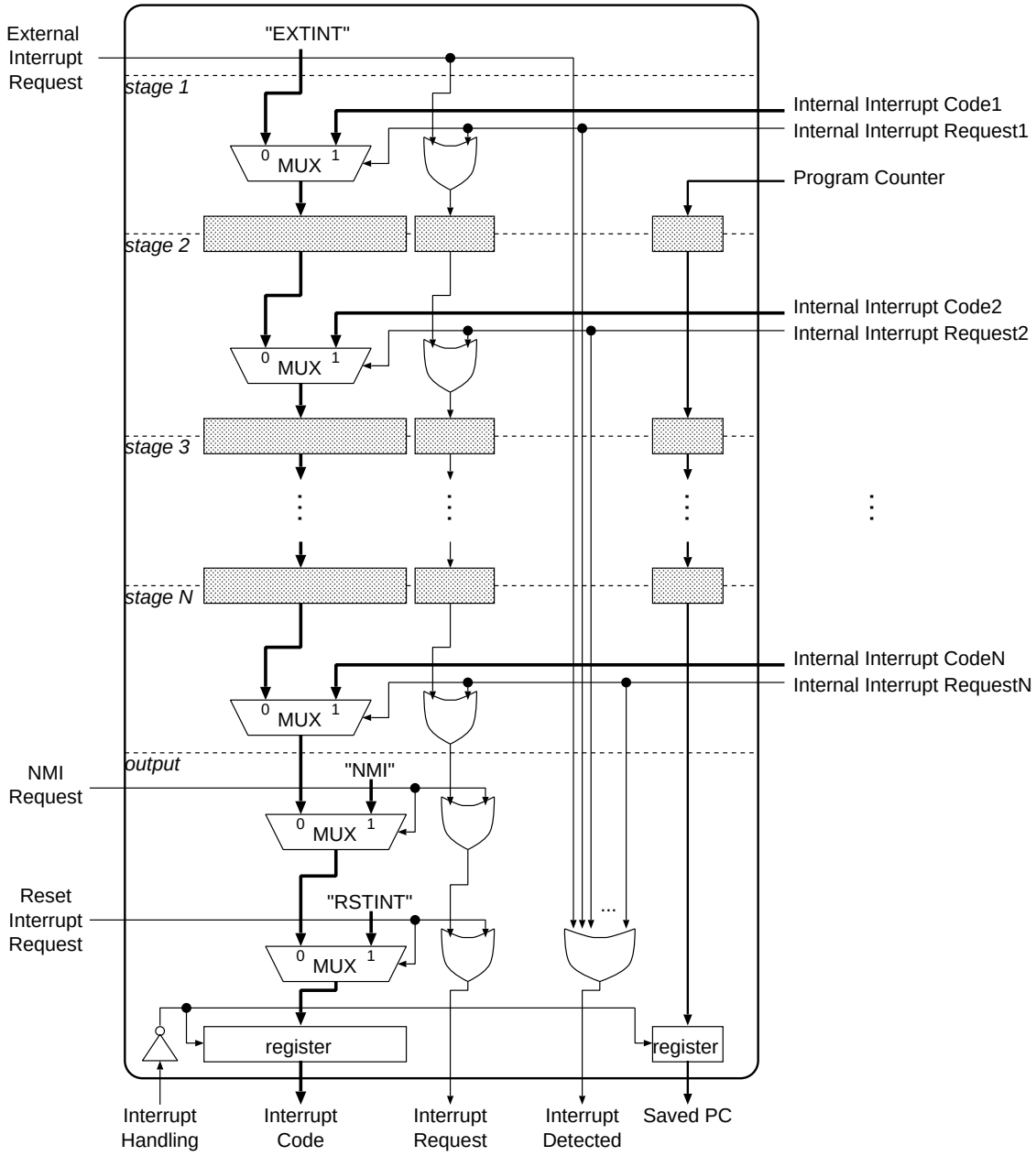


Figure 3.12: Hardware model of interrupt pipeline to handle a nonmaskable interrupt

$Detected_{stage,intr}$, a detection signal of $intr$ in $stage$, is represented as follows:

$$Detected_{stg,intr} = valid_{stg} \wedge \left\{ \bigvee_{ope \in Ope_{intr}} Dec_{ope} \wedge cond_{ope,intr} \right\}, \tag{3.12}$$

where Dec_{ope} is a decode signal of operation ope, and $cond_{ope,intr}$ represents that $intr$ occurs in ope.

Let $Intr_{stg}$ be interrupts that occur in stage stg. Then, $IntrReq_{stg}$, a signal to indicate that any interrupt occurs in pipeline stage stg, is described using $Detected_{stg,intr}$ as follows:

$$IntrReq_{stg} = \bigvee_{intr \in Intr_{stg}} Detected_{stg,intr}. \tag{3.13}$$

3.5.4.3 Internal interrupt model of VLIW processors

In VLIW processors, it is needed to select one interrupt occurring in the higher-prioritized slot among multiple interrupts in a stage. $Detected_{stg,intr,slot}$, a signal to detect internal interrupt $intr$ occurring in $slot$ in stage stg, is described as follows:

$$Detected_{stg,intr,slot} = valid_{stg} \wedge \left\{ \bigvee_{\substack{ope \in Ope_{intr} \\ rg \in Rg_{slot}}} Dec_{rg,ope} \wedge Active_{rg} \wedge cond_{ope,intr} \right\}. \tag{3.14}$$

$Detected_{stg,slot}$, a signal to detect an interrupt occurring in $slot$ in stg, is described as follows:

$$Detected_{stg,slot} = \bigvee_{intr \in Intr_{stg}} Detected_{stg,intr,slot}. \tag{3.15}$$

Therefore, using these signals, $IntrCode_{stg}$, an internal interrupt identifier in stage stg, is represented as follows:

$$IntrCode_{stg} = \bigvee_{slot \in Slot} \left\{ \bigwedge_{s > slot} \left\{ \overline{Detected_{stg,s}} \right\} \wedge \bigvee_{intr \in Intr} \{ Code_{intr} \wedge Detected_{stg,intr,slot} \} \right\}, \tag{3.16}$$

where $s > slot$ means that slot s has higher priority than slot $slot$.

$IntrReq_{stg}$, a signal to indicate that any interrupt occurs in pipeline stage stg, is represented using $Detected_{stg,intr,slot}$ as follows:

$$IntrReq_{stg} = \bigvee_{intr \in Intr_{stg}} Detected_{stg,intr,slot}. \tag{3.17}$$

The model of [45] saves the value of program counter when an interrupt occurs. Similarly, the interrupt pipeline in VLIW processors saves a slot number in which an interrupt occurs as well as the value of program counter, in order to allow an interrupt handler to precisely distinguish the operation that causes the interrupt.

Table 3.1: A VLIW pattern table of the VLIW processor in the preliminary experiment.

VLIW Pattern	Slot1	Slot2	Slot3
#1	OG_{ALU}	OG_{ALU}	OG_{NOP}
#2	OG_{NOP}	OG_{ALU}	OG_{ALU}

3.6 Generation method for efficient VLIW processors

In this section, a resource group assignment method of the VLIW processor generation method proposed in Section 3.5 is discussed furthermore. First, a relation between FU allocation and design quality, such as area and delay time, is examined. Then, the importance of FU allocation to generate a fine quality VLIW processor is discusses.

3.6.1 Relation between FU allocation and design quality

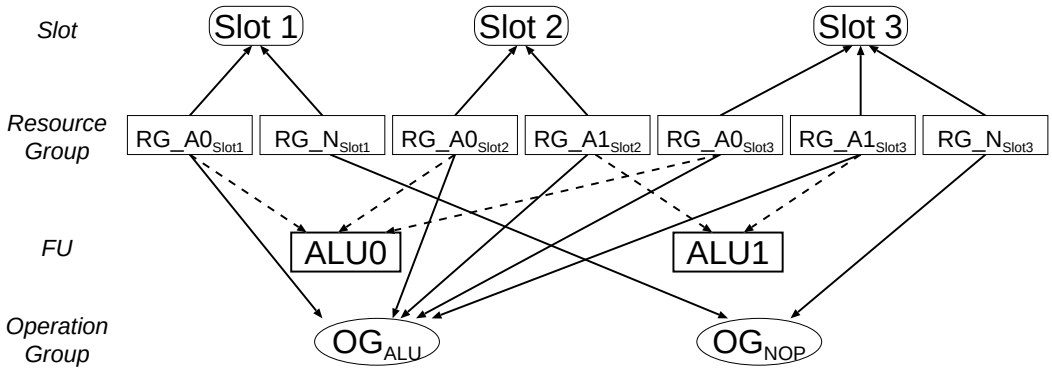
In the resource group decision algorithm explained in Section 3.5.2, even if there are multiple candidates that can be assigned to a slot, a resource group that is found first is adopted. No matter which candidate is adopted, a generated VLIW processor properly works. However, the design quality might change significantly depending on the adopted candidate. In this section, a relation between the design quality, that is area and delay time, and FU allocation is examined.

3.6.1.1 Preliminary experiment

Simple 3-slot VLIW processors that have only two operation groups, OG_{ALU} and OG_{NOP}, are designed. Figure 3.13 shows a dispatch table of this VLIW processor. This processor supports resource groups, RG_A0_{Slot1}, RG_A0_{Slot2}, RG_A1_{Slot2}, RG_A0_{Slot3}, and RG_A1_{Slot3} for operating OG_{ALU}. It supports for operating OG_{NOP} RG_N_{Slot1} and RG_N_{Slot3}. Furthermore, RG_A0_{Slot1}, RG_A0_{Slot2}, and RG_A0_{Slot3} include ALU0, and RG_A1_{Slot2} and RG_A1_{Slot3} include ALU1. VLIW patterns for these processors are shown in Table 3.1.

In this case, there are two candidates of possible resource group assignment, as shown in Table 3.2 and Table 3.3. In resource group assignment 1, Slot 2 uses ALU0 in VLIW pattern #1, while it uses ALU1 in VLIW pattern #2, as shown in Table 3.2, On the other hand, in resource group assignment 2, every slot uses either ALU0 or ALU1 through all VLIW patterns, as shown in Table 3.3,

Both of the processors will work properly, however, the design quality might be different.



(a) Dispatch Model

```
VLIW_pattern {
  { Slot1: OG_ALU, Slot2: OG_ALU, Slot3: OG_NOP },
  { Slot1: OG_NOP, Slot2: OG_ALU, Slot3: OG_ALU },
};
opegroup_resgroup {
  OG_ALU: RG_A0_Slot1, RG_A0_Slot2, RG_A1_Slot2, RG_A0_Slot3, RG_A1_Slot3;
  OG_NOP: RG_N_Slot1, RG_N_Slot3;
};
slot_resgroup {
  Slot1: RG_A0_Slot1, RG_N_Slot1;
  Slot2: RG_A0_Slot2, RG_A1_Slot2;
  Slot3: RG_A0_Slot3, RG_A1_Slot3, RG_N_Slot3;
};
```

(b) Dispatch Rule Description

Figure 3.13: A dispatch table of the VLIW processor in the preliminary experiment.

Table 3.2: Resource group assignment 1.

VLIW Pattern	Slot1	Slot2	Slot3
#1	$RG_{A0_{Slot1}}$	$RG_{A1_{Slot2}}$	$RG_{N_{Slot3}}$
#2	$RG_{N_{Slot1}}$	$RG_{A0_{Slot2}}$	$RG_{A1_{Slot3}}$

Table 3.3: Resource group assignment 2.

VLIW Pattern	Slot1	Slot2	Slot3
#1	$RG_{A0_{Slot1}}$	$RG_{A1_{Slot2}}$	$RG_{N_{Slot3}}$
#2	$RG_{N_{Slot1}}$	$RG_{A1_{Slot2}}$	$RG_{A0_{Slot3}}$

Table 3.4: Synthesis results of the preliminary experiment.

			Assign. 1	Assign. 2	(Reduction)	
Entire Processor	Area (gate)	Combinational	43006	**42278**	728	(1.7%)
		Non-Combinational	14502	14483	19	(0.0%)
		Total	57512	**56765**	747	(1.3%)
	Max Delay (ns)		12.65	**12.26**	0.39	(3.1%)
Data-path Selectors	# of selectors		18	**16**	2	(11.1%)
	Area (gate)	Total	2914	**2339**	575	(19.7%)

(0.14 CMOS Library)

3.6.1.2 Synthesis results and discussion of the preliminary experiment

Table 3.4 shows synthesis results of the VLIW processors. Area and delay time were measured using Synopsys Design Compiler with $0.14\mu m$ CMOS standard cell library. Figure 3.14 illustrates ALU0 and ALU1, and related data-path selectors and pipeline registers, in the architecture of the VLIW processors. Table 3.4 shows that the processor with resource group assignment 2 has fewer data-path selectors, and lower area of data-path selectors and the entire processor. The reason is that the required number and size of data-path selectors are decreased since ALU1 is only used by Slot2, and Slot2 only uses ALU1 in assignment 2, while ALU1 is used by two slots, and Slot2 uses both of ALU0 and ALU1 in assignment 1, as shown in Fig. 3.14.

The experimental results show that allocating fewer FUs to a slot or being allocated fewer slots for an FU improves design quality in terms of area and delay time. The reason is that the number of input ports of data-path selector placed on the input port of FU decreases while the number of slots to be allocated to an FU decreases, and also the number of input ports of data-path selector placed on the input port of the register file decreases while the number of FUs to be allocate to a slot decreases. Consequently, in case of multiple candidates for a VLIW pattern, a candidate that contains fewer FUs for a slot would be beneficial in terms of the design quality.

3.6.2 Efficient resource group assignment method

As was mentioned in Section 3.6.1, trying to allocate fewer FUs for a slot and trying to be allocated fewer slots for an FU give a fine quality VLIW processor. This section describes a resource group assignment method that can generate a fine quality VLIW processor.

Let n be the number of instances of FU that is required for operations in operation group og. First, an FU allocation method in a case that a VLIW processor can issue n operations of og at a time from all slots is discussed. Then, a resource group assignment method to realize the FU allocation method is explained. In this section, cases of three, four, and five slots, that are moderate cases to find a trend of optimal FU allocation, are discussed.

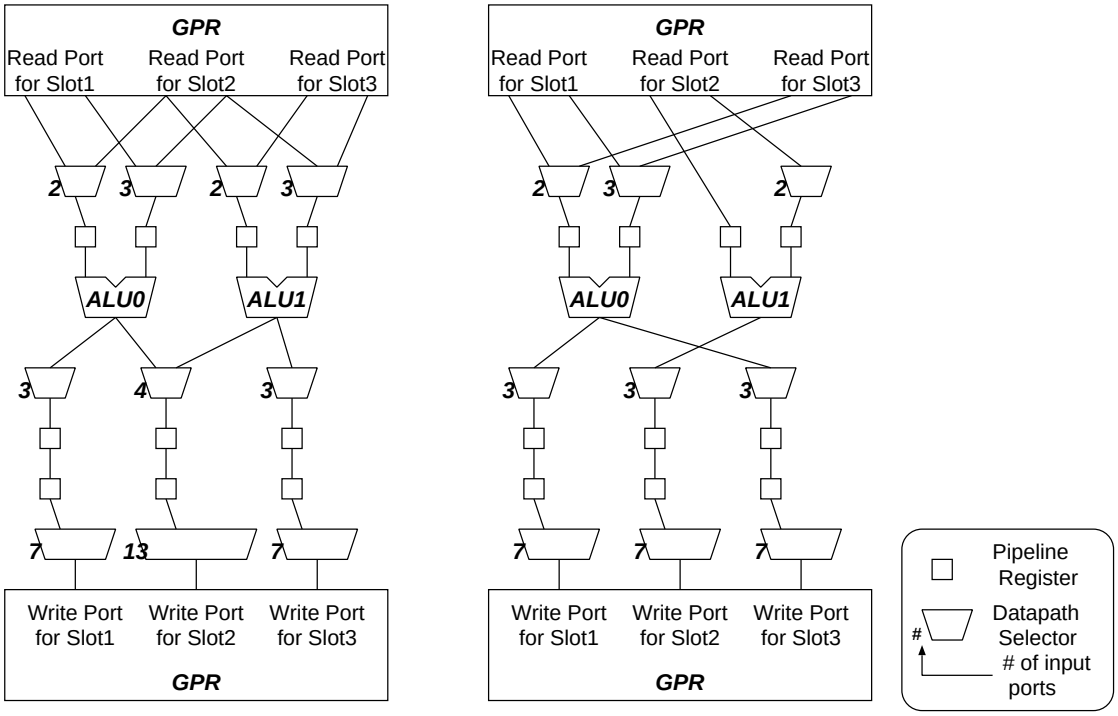


Figure 3.14: Architecture of VLIW processors with different resource group assignment.

3.6.2.1 Case of 3 Slots

If a VLIW processor has three slots, when n is three or one, allocation between slots and FUs is determined. Then, in this section, only a case of $n = 2$ is discussed.

Figure 3.15 shows allocation of two FUs to three slots. In Fig. 3.15, an operation in Slot1 is operated on FU_A, an operation in Slot2 is operated on FU_A or FU_B, and an operation in Slot3 is operated on FU_B. This allocation can operate all combinations of og's operation from all slots.

3.6.2.2 Case of 4 Slots

If a VLIW processor has four slots, when n is four or one, allocation between slots and FUs is decided. Then, in this section, cases of $n = 2$ or $n = 3$ are explained.

Figure 3.16 shows allocation of two FUs to four slots. In Fig. 3.16, an operation in Slot1 is operated on FU_A, an operation in Slot2 and Slot3 is operated on FU_A or FU_B, and an operation in Slot4 is operated on FU_B. This allocation can operate all combinations of og's operation from all slots.

Figure 3.17 shows allocation of three FUs to four slots. In Fig. 3.17, an operation in Slot1 is operated on FU_A, an operation in Slot2 is operated on FU_A or FU_B, an operation in Slot3 is operated on FU_B or FU_C, and an operation in Slot4 is operated on FU_C. This allocation

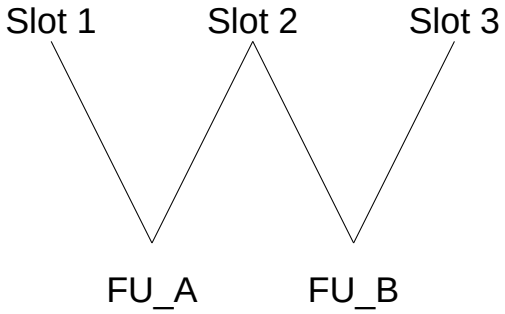


Figure 3.15: Allocation of two FUs to three slots.

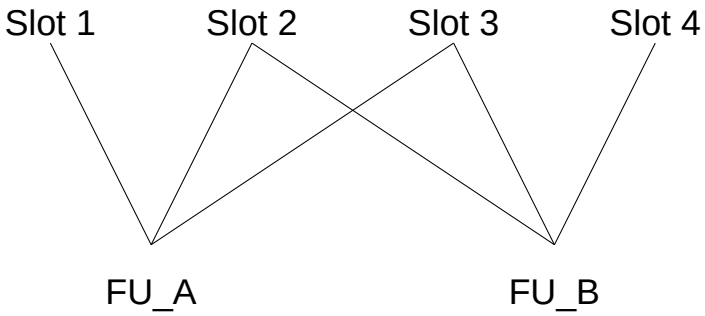


Figure 3.16: Allocation of two FUs to four slots.

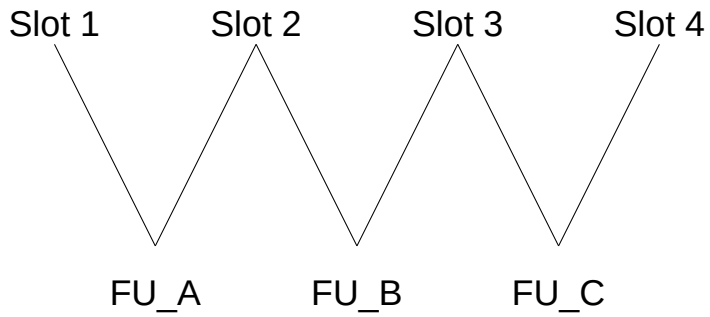


Figure 3.17: Allocation of three FUs to four slots.

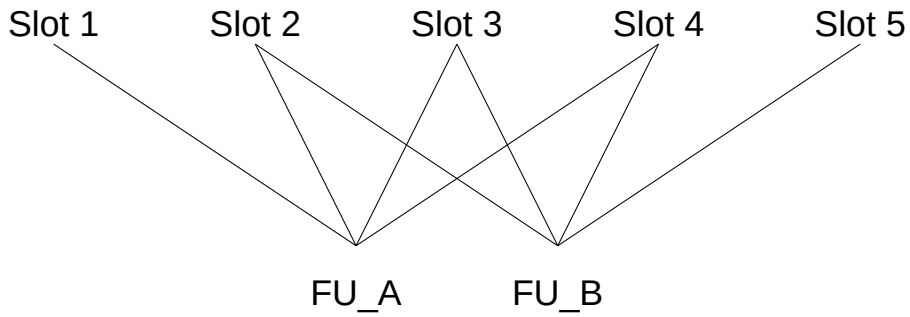


Figure 3.18: Allocation of two FUs to five slots.

can operate all combinations of og's operation from all slots.

3.6.2.3 Case of 5 Slots

If a VLIW processor has five slots, when n is five or one, allocation between slots and FUs is decided. Then, in this section, cases of two, three or four FUs are explained.

Figure 3.18 shows allocation of two FUs to five slots. In Fig. 3.18, an operation in Slot1 is operated on FU FU_A, an operation in Slot2, Slot3 and Slot4 is operated on FU_A or FU_B, and an operation in Slot5 is operated on FU_B. This allocation can operate all combinations of og's operation from all slots.

Figure 3.19 shows allocation of three FUs to five slots. In Fig. 3.19, an operation in Slot1 is operated on FU_A, an operation in Slot2 is operated on FU_A or FU_B, an operation in Slot3 is operated on FU_A, FU_B or FU_C, an operation in Slot4 is operated on FU_B or FU_C, and an operation in Slot5 is operated on FU_C. This allocation can operate all combinations of og's operation from all slots.

Figure 3.20 shows allocation of four FUs to five slots. In Fig. 3.20, an operation in Slot1 is operated on FU_A, an operation in Slot2 is operated on FU_A or FU_B, an operation in Slot3 is operated on FU_B or FU_C, an operation in Slot4 is operated on FU_C or FU_D, and an

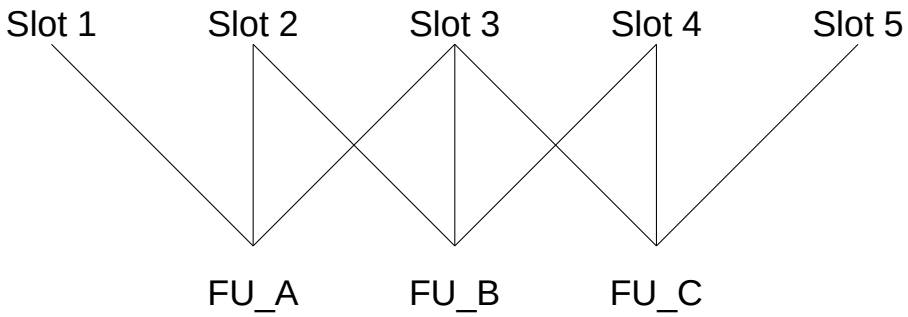


Figure 3.19: Allocation of three FUs to five slots.

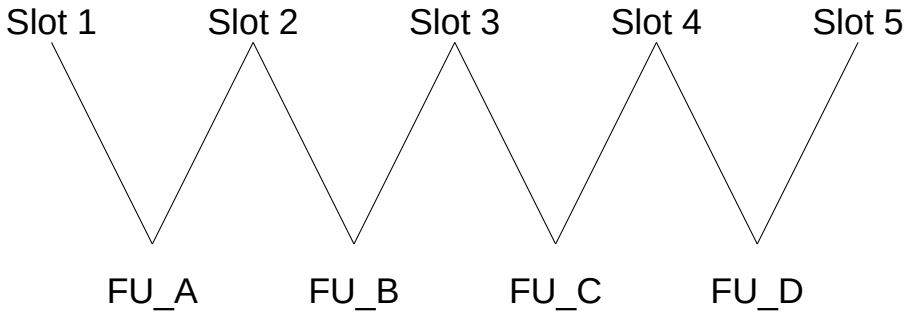


Figure 3.20: Allocation of four FUs to five slots.

operation in Slot5 is operated on FU_D. This allocation can operate all combinations of og's operation from all slots.

3.6.2.4 An FU Allocation Algorithm

Based on the discussion from section 3.6.2.1 to section 3.6.2.3, an FU allocation algorithm is described. Let x be an FU allocation, $Slot$ be a set of slots, FU be a set of FUs, $f_s(x)$ be the number of FUs that is allocated with slot s in allocation x, $s_f(x)$ be the number of slots that is allocated by FU f in allocation x. $F(x)$, the total of $f_s(x)$ for all slots, and $S(x)$, the total of $s_f(x)$ for all FUs are represented as follows:

$$F(x) = \sum_{s \in Slot} f_s(x) \tag{3.18}$$

$$S(x) = \sum_{f \in FU} s_f(x) \tag{3.19}$$

Then, the proposed FU allocation algorithm selects an FU allocation candidate to minimize $(F(x) + S(x))$.

3.6.2.5 Resource Group Assignment Algorithm

In the VLIW processor generation method explained in Section 3.5, the algorithm chooses a resource group only from given resource groups specified in the input specification resource groups that can operate a VLIW pattern are selected in given resource groups as an input specification. In other words, resource group assignment or FU allocation can be controlled with the input specification. Consequently, it is possible to assign appropriate resource groups to realize FU allocation mentioned before, by describing as follows:

1. For operation group og, obtain FU_{og}, which is a set of FUs that can process og.
2. Let $Slot_{og}$ be a set of slots that issues og. According to the discussion described in section 3.6.2.4, obtain FU_{slot}, which is assignment of FU_{og} to $Slot_{og}$.
3. For $\exists slot \in Slot_{og}$, create resource groups that include FUs $fu \in FU_{slot}$.

Table 3.5: Parameters of designed VLIW processors.

Parameter	Value
# of Slots	1, 2, 3, 4
# of Pipeline Stages	3, 4, 5
Instruction Bit Width	32, 24
# of VLIW patterns	5 – 938
# of ALUs	1, 2, 3, 4
# of Multipliers	1, 2, 3
# of Dividers	0, 1
# of Shifters	1, 2, 3, 4
Interrupts	none

3.7 Experimental Results and Discussion

In this section, the proposed VLIW processor generation method is evaluated.

3.7.1 Evaluation of VLIW processor generation method

In order to confirm feasibility of the proposed VLIW processor generation method, 36 VLIW processors are designed using the proposed approach. In this section, the detail of the experiment and its considerations are discussed.

3.7.1.1 Experimental setup

36 processors are designed and processor specification descriptions are created for each processor. These processors have the different number of slots and FUs and various type of dispatching rules. Then HDL description is generated for each specification description using the implemented processor generation method.

Hardware are and maximum delay time of the generated processors are evaluated after logic synthesis. Switching information of processor is obtained from gate level simulation with applications and power consumption is estimated with the information.

The HDL description was generated on Intel Pentium4 2.8GHz, 512MB memory, and RedHat Linux 7.3. Area, delay time, and power consumption were estimated using Synopsys Design Compiler with 0.14μm CMOS library.

3.7.1.2 Experimental results

36 VLIW processors with up to 4 slots are designed. They vary in dispatching rule, FU, the number of pipeline stages, and instruction width, as shown in Table 3.5. Since so-called copy and paste technique can be used to describe specification of derivatives, it took only eight

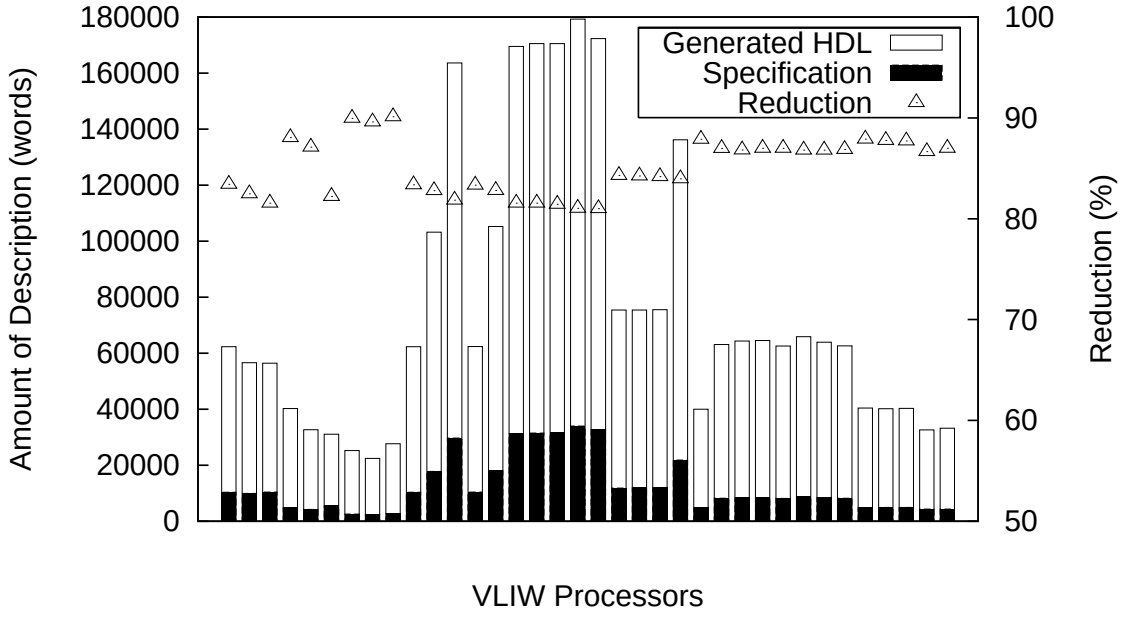


Figure 3.21: Reduction of the amount of description.

hours to create 36 specifications. Furthermore, generation time of HDL description from the specification was from 2 seconds to 15 seconds.

On the other hand, there was the tremendous reduction of the amount of description that a designer has to describe. Figure 3.21 shows a comparison of the amount of description between the processor specification description and generated HDL description. In this figure, the amount of description is counted by the number of words, since it is assumed that the number of words represents the complexity of description better than the number of lines. Since a line is sometimes too long, it is not fair to compare the complexity by the number of lines. In Fig. 3.21, x axis represents the generated 36 processors. A white and a black bar represent the number of words in generated HDL description and in the processor specification description for each processor, respectively. A triangle represents the percentage of the processor specification description over the HDL description (refer to the right y axis). The percentage of the proposed specification description over the VHDL description is from 11% to 22%, and the average percentage is only 18%. This result shows that designers describe a specification in the proposed method 82% less than HDL description; the amount of the specification that designers have to describe by hand is about 5 times smaller than that of HDL description.

A RISC processor generated by the scalar processor generation method has only 20% larger area than manually designed HDL description [48]. Though a generated VLIW processor has not been compared with a manually designed VLIW processor yet, it is assumed that the quality of generated HDL description is almost the same as that of manually designed HDL description.

Figure 3.22 shows a trade-off between area and performance of the generated processors. The x axis denotes the area of generated processors, and the y axis denotes the execution time with an FIR filter application in maximum frequency. In Fig. 3.22, processors from A to F are

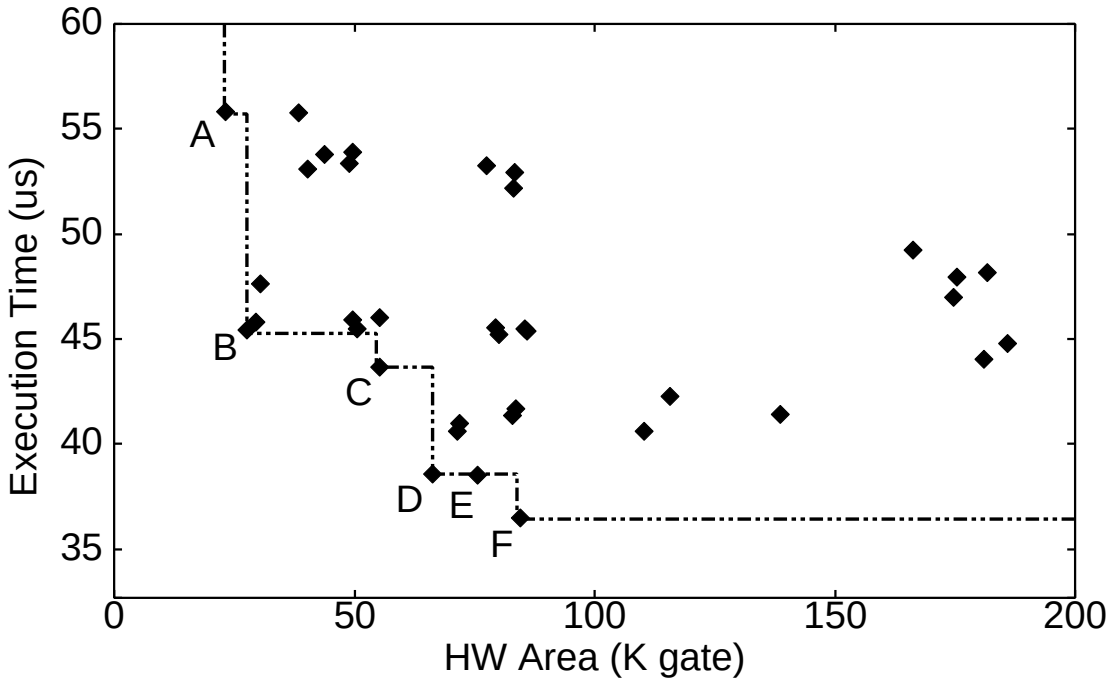


Figure 3.22: Trade-off between HW area and execution time of FIR filter application.

candidates of design space exploration. This result shows that exploration of large design is possible using the proposed method, and designers can easily find architecture candidates from the huge design space.

3.7.2 Evaluation of efficient VLIW processor generation method

In this section, a generation method of efficient VLIW processor described in Section 3.6 is evaluated.

3.7.2.1 Experimental Procedure and Environment

The experiment has been conducted in the procedure as follows.

1. Design a VLIW processor with the proposed resource group assignment method.
2. Design eleven VLIW processors with different resource group assignment, and create their processor specification descriptions.
3. Generate HDL description from the descriptions by the implemented VLIW processor generation method.
4. Measure hardware area and maximum delay time with logic synthesis.

Area and delay time were measured using Synopsys Design Compiler with $0.14\mu m$ CMOS library.

3.7.2.2 Designed VLIW Processors

This section shows the specification of designed VLIW processors. The parameters of processors designed for this experiment is shown below.

- Operation width: 32bit
- The number of operations: 57
- The number of registers in register file: 32
- The number of register file: 1
- The number of slots: 4
- The number of pipeline stages: 5
- The number of ALU: 3
- The number of multiplier: 2
  The multiplication algorithm is sequential type.
- The number of divider: 2
  The division algorithm is sequential type.
- The number of shifter: 3
- The number of data memory access units: 1
- Data forwarding: N/A

The instruction set of designed processors is based on the DLX architecture [49], and every operation can be issued from all slots (e.g. homogeneous VLIW processors). However, the maximum number of parallel executable operations depends on the number of FUs; for example, up to three shift operations can be issued at a time. Table 3.6 shows operations in the instruction set, their operation group, and their required FUs. With the limitation of the maximum number of parallel executable operations, 1565 VLIW patterns has been created.

Table 3.7 shows FU allocation of designed VLIW processors. "the proposed method" processor is a VLIW processor generated using the proposed resource group assignment method. Each entry of Table 3.7 includes $f_s(x)$ and $s_f(x)$ that are explained in section 3.6.2.4. For instance, in assignment 1 VLIW processor, $ALU_s(x)$ ($f_s(x)$ for ALU) is $\{1, 2, 3, 3\}$. This means that the numbers of ALUs allocated with each slot of the processor are 1, 2, 3, and 3, respectively. Moreover, $s_{ALU}(x)$ ($s_f(x)$ for ALU) is $\{4, 3, 2\}$. This means that the numbers of slots allocated to each ALU of the processor are 4, 3, and 2, respectively. Therefore, FU allocation for ALUs of assignment 1 VLIW processor is shown in Fig. 3.23.

3.7.2.3 Experimental Results

In this section, experimental results are shown.

Table 3.6: Instruction set of designed VLIW processors.

Operation Group Name	Operation Name	Required FU
OG_SFT	SLL, SRL, SRA, SLLI, SRLI, SRAI	Shifter
OG_ALU	ADD, ADDU, ADDI, ADDUI, SUB, SUBU, SUBI, SUBUI, AND, ANDI, OR, ORI, XOR, XORI, SLT, SGT, SLE, SGE, SEQ, SNE, SLTI, SGTI, SLEI, SGEI, SEQI, SNEI, SLTU, SGTU, SLEU, SGEU	ALU
OG_JMP	BEQZ, BNEZ, J, JAL, JR, JALR	ALU, Sign extender
OG_MEM	LB, LH, LW, LBU, LHU, SB, SH, SW	ALU, Sign extender
OG_MUL	MULT, MULTU	Multiplier
OG_DIV	DIV, DIVU, MOD, MODU	Divider
OG_LHI	LHI	

Table 3.7: FU allocation of designed VLIW processors.

VLIW processor ID	for ALU and shifter		for multiplier and divider	
	$f_s(x)$	$s_f(x)$	$f_s(x)$	$s_f(x)$
the proposed method	{1, 2, 2, 1}	{2, 2, 2}	{1, 2, 2, 1}	{3, 3}
assignment 1	{1, 2, 3, 3}	{4, 3, 2}	{1, 2, 2, 2}	{4, 3}
assignment 2	{1, 2, 3, 1}	{3, 2, 2}	{1, 2, 2, 2}	{4, 3}
assignment 3	{1, 2, 2, 2}	{3, 2, 2}	{1, 2, 2, 2}	{4, 3}
assignment 4	{1, 2, 3, 2}	{4, 2, 2}	{1, 2, 2, 2}	{4, 3}
assignment 5	{1, 2, 3, 2}	{3, 3, 2}	{1, 2, 2, 2}	{4, 3}
assignment 6	{1, 2, 3, 3}	{4, 3, 2}	{1, 2, 2, 1}	{3, 3}
assignment 7	{1, 2, 2, 1}	{2, 2, 2}	{1, 2, 2, 2}	{4, 3}
assignment 8	{1, 2, 3, 1}	{3, 2, 2}	{1, 2, 2, 1}	{3, 3}
assignment 9	{1, 2, 2, 2}	{3, 2 ,2}	{1, 2, 2, 1}	{3, 3}
assignment 10	{1, 2, 3, 2}	{4, 2, 2}	{1, 2, 2, 1}	{3, 3}
assignment 11	{1, 2, 3, 2}	{3, 3, 2}	{1, 2, 2, 1}	{3, 3}

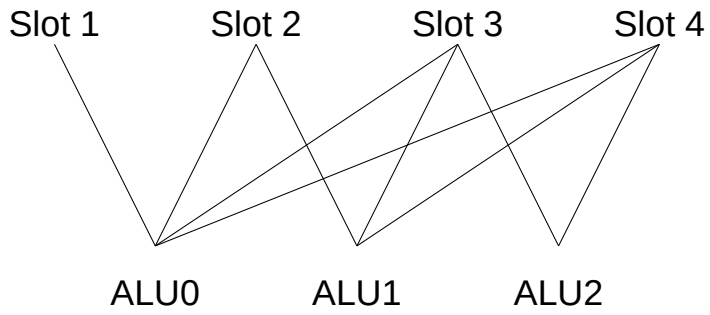


Figure 3.23: FU allocation for ALUs in assignment 1.

Table 3.8: Comparison of area and delay between designed VLIW processors.

	Entire Processor		Datapath Selectors	Controller
VLIW processor ID	Area (gate)	Delay (ns)	Total Area (gate)	Area (gate)
the proposed method	93170	14.9	9662	8802
assignment 1	95536	15.6	11320	10167
assignment 2	93864	15.0	10199	9150
assignment 3	94172	15.3	10142	9807
assignment 4	94605	14.5	10733	9709
assignment 5	94788	15.1	10758	10127
assignment 6	94942	15.2	10943	9995
assignment 7	93530	15.4	10008	8926
assignment 8	93556	15.4	9882	8928
assignment 9	93942	14.6	9844	9478
assignment 10	94237	15.4	10429	9373
assignment 11	94219	14.7	10410	9838

Figure 3.24 illustrates a comparison of hardware area among the VLIW processors. Figure 3.24 represents that hardware area differs among the different FU allocation and the proposed method that uses an efficient FU allocation achieves the smallest hardware area. Table 3.8 shows the detailed results of VLIW processors with different FU allocation. In Table 3.8, the area and delay time of the entire processor are shown as well as the area of data-path selectors and a controller. Note that the controller is a part of VLIW processor which decodes an instruction and dispatches operations, and it also controls the pipeline status such as pipeline interlock. Table 3.8 shows that the area reduction is mainly come from the reduction of data-path selectors and controller.

3.7.2.4 Discussion

The change of resource group assignment leads to the change of FU allocation, then the size of required data-path selectors also changes.

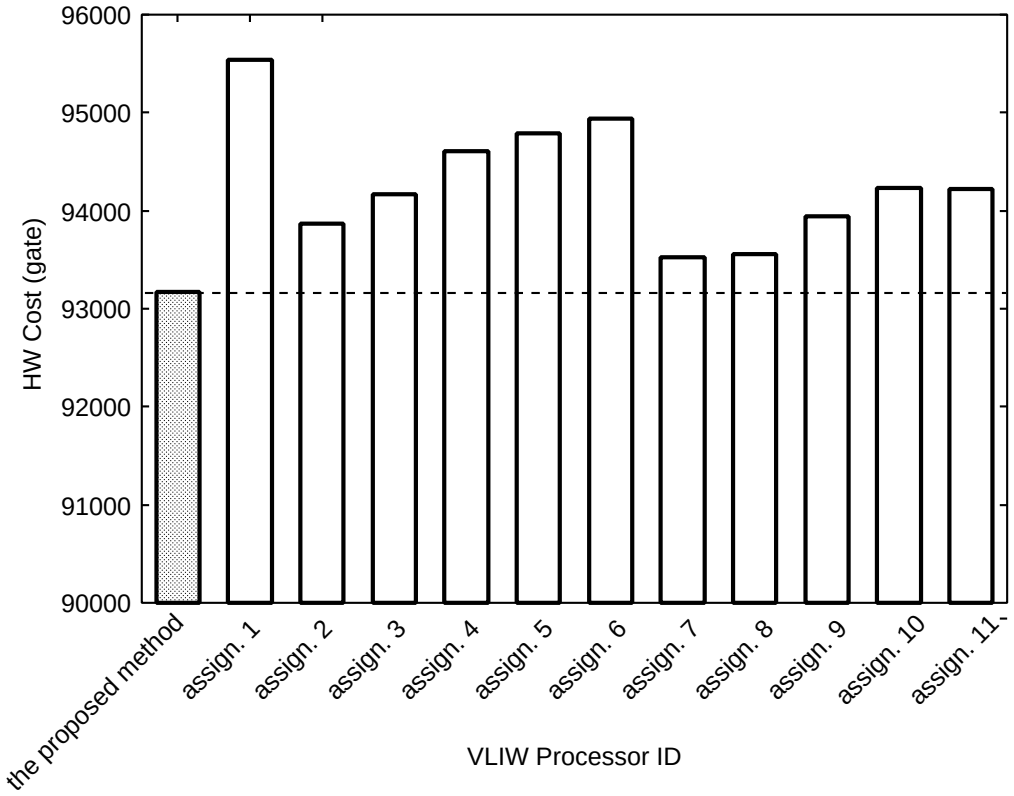


Figure 3.24: Comparison of hardware area.

Table 3.9 explains allocation of two FUs to four slots for each VLIW pattern. In this experiments, a case of two FUs is a case of multipliers or dividers. In Table 3.9, "*" indicates a slot that issues an operation using the FU, in each VLIW pattern. The column of assignment 1 and the proposed method in Table 3.9 represent an FU allocated to each slot for each VLIW pattern. According to Table 3.9, only one FU, FU1, is allocated to Slot4 in the proposed assignment method, while two FUs, FU0 and FU1, are allocated to Slot4 in assignment 1.

Table 3.10 explains allocation of three FUs to four slots for each VLIW pattern. In this experiments, a case of three FUs is a case of ALUs or shifters. According to Table 3.10, only one FU, FU2, is allocated to Slot4 and two FUs, FU1 and FU2, are allocated to Slot3 in the proposed method, while three FUs, FU0, FU1 and FU2, are allocated to Slot3 and Slot4 in assignment 1.

Allocating fewer FUs to a slot makes the size of data-path selectors that choose input data of a register file smaller. Moreover, since it also leads to allocating an FU to fewer slots the size of data-path selectors that choose input data of an FU also becomes smaller. Since fewer data-path selectors benefit area and delay time, fine quality VLIW processors can be generated using the proposed method. Consequently, the proposed resource group assignment method is effective to generate a VLIW processor with small area and delay time.

Table 3.9: Allocation of two FUs to four slots for each VLIW pattern.

VLIW Pattern				Assignment 1				the proposed method			
S1	S2	S3	S4	S1	S2	S3	S4	S1	S2	S3	S4
*				FU0				FU0			
	*				FU0				FU0		
		*				FU0				FU0	
			*				FU0				FU1
*	*			FU0	FU1			FU0	FU1		
*		*		FU0		FU1		FU0		FU1	
*			*	FU0			FU1	FU0			FU1
	*	*			FU0	FU1			FU0	FU1	
	*		*		FU0		FU1		FU0		FU1
		*	*			FU0	FU1			FU0	FU1

Table 3.10: Allocation of three FUs to four slots for each VLIW pattern.

VLIW Pattern				Assignment 1				the proposed method			
S1	S2	S3	S4	S1	S2	S3	S4	S1	S2	S3	S4
*				FU0				FU0			
	*				FU0				FU0		
		*				FU0				FU1	
			*				FU0				FU2
*	*			FU0	FU1			FU0	FU1		
*		*		FU0		FU1		FU0		FU1	
*			*	FU0			FU1	FU0			FU2
	*	*			FU0	FU1			FU0	FU1	
	*		*		FU0		FU1		FU0		FU2
		*	*			FU0	FU1			FU1	FU2
*	*	*		FU0	FU1	FU2		FU0	FU1	FU2	
*	*		*	FU0	FU1		FU2	FU0	FU1		FU2
*		*	*	FU0		FU1	FU2	FU0		FU1	FU2
	*	*	*		FU0	FU1	FU2		FU0	FU1	FU2

Table 3.11: Occurrence conditions of added interrupts.

Interrupt Name	Occurrence Condition
Nonmaskable interrupt	when the value of “NMI” input port becomes active.
Overflow interrupt	when the overflow flag of ALU becomes active in operation of operation group OG_ALU
Zero division interrupt	when the error flag of divider becomes active in operation of operation group OG_DIV

3.7.3 Evaluation of VLIW processor generation method with interrupt model

This section describes an experiment that is performed in order to confirm the feasibility of the proposed interrupt model of VLIW processor.

The experiment has been conducted in the procedure as shown below.

1. Design a VLIW processor with interrupt features based on the proposed interrupt model, and create its processor specification description.
2. Generate HDL from the description by the implemented VLIW processor generation system.
3. Simulate the VLIW processor to confirm whether or not it can process interrupts.

3.7.3.1 Designed Processor

A designed VLIW processor for this experiment is based on the VLIW processor explained in Section 3.7.2, and interrupts are added as shown below. Moreover, *NMI* input port of the processor core is also added to realize a nonmaskable interrupt.

- Nonmaskable interrupt
- Two kind of internal interrupt
  - Overflow interrupt
  - Zero division interrupt

Table 3.11 shows interrupts and their occurrence condition. A nonmaskable interrupt occurs when the value of *NMI* port of the processor becomes active. An overflow interrupt occurs when an ALU asserts overflow flag in an arithmetic operation. A zero division interrupt occurs when divider outputs error flag in a division operation. Furthermore, in case of multiple interrupts, an interrupt occurred from the least number of slot is processing.

Table 3.12 shows behavior of added interrupts. In Table 3.12, *PC* means a program counter. In the designed processor, a nonmaskable interrupt, an overflow interrupt, and a zero division interrupt invoke interrupt handlers located on the address of ”0x00008800”, ”0x00008000”, ”0x00008400”, respectively.

Table 3.12: Behavior of added interrupts.

Interrupt Name	Behavior
Nonmaskable interrupt	PC ← 0x00008800, reset a register file and an instruction register.
Overflow interrupt	PC ← 0x00008000.
Zero division interrupt	PC ← 0x00008400.

Table 3.13: Comparison of area and delay between VLIW processors with and without interrupts.

VLIW Processor	Area (gate)	Max Delay (ns)
with Interrupts	94329	14.8
w/o Interrupts	93095	14.2

3.7.3.2 Evaluating of Generated Processor

In this section, evaluation of the generated processor is describes.

The behavior of a nonmaskable interrupt and internal interrupts is checked through the RTL simulation. It is confirmed that the generated processor works properly in case that the nonmaskable interrupt or internal interrupts occur, including the case of multiple internal interrupts.

Design Quality of Interrupt Handling Circuits Table 3.13 shows comparison of hardware area and maximum delay time between the VLIW processors with interrupts and a VLIW processor without interrupts. According to Table 3.13, increases of area and delay time are one percent and four percent, respectively. This result can be considered as appropriate increases.

3.7.3.3 Discussion

This section evaluated a VLIW interrupt model that can handle a nonmaskable interrupt as well as an internal interrupt. In a VLIW processor, though multiple internal interrupts may occur in a pipeline stage at a time, the proposed model can select an appropriate interrupt according to the slot priority. The experimental results show that all interrupts based on the proposed interrupt model for VLIW processor works properly.

3.8 Conclusion

This chapter proposed a generation method based on the configurable VLIW processor model [44]. In the proposed method, pipeline stages, slots and dispatching rule are configurable, and pipeline control logic is generated automatically. In the design of pipeline processors, designing pipeline control logic is a troublesome and difficult part, however, automatic generation of such logic helps designers to concentrate on customizing the processor architecture. Experimental

results shows that the proposed method can generate a VLIW processor from a high level description, which is 80% to 90% smaller than HDL description, as shown in Section 3.7.1. And also, the generation time of HDL description is sufficiently short, that is from 2 to 15 seconds.

This chapter also proposed a resource group assignment algorithm to generate VLIW processors of smaller area and shorter delay time. The proposed assignment algorithm minimizes a total of the number of slots that an FU is allocated with and the number of FUs that are allocated with a slot. The experimental results indicated that the proposed algorithm can generate fine-quality VLIW processors.

Though a generated VLIW processor has not been compared with a manually designed VLIW processor yet, it is assumed that the quality of generated HDL description is almost the same as that of manually designed HDL description as discussed in Section 3.7.1.2.

Since the specification description supports a wide range of dispatching rules and the amount of description is sufficiently small, it is possible to generate a wide range of fine-quality VLIW processors in a short time. Note that a simple *copy and paste* strategy can be employed during preparation of the processor specification description. Hence, the actual effort that designers have to describe is much smaller than the manual design of HDL. Therefore, the proposed method can significantly improve the design productivity of VLIW processors.

Chapter 4

Operation shuffling algorithm for low energy L0 cluster

The method described in Chapter 3 significantly improves the design productivity of VLIW processors. The chapter also gives an algorithm to make selectors in a VLIW processor smaller, which is also beneficial for power reduction.

VLIW processors, however, have a power bottleneck in the instruction memory hierarchy. Therefore, energy reduction on the instruction memory hierarchy is the next challenge for VLIW processors. This chapter describes an approach to reduce the energy consumption in the instruction memory hierarchy using an operation shuffling algorithm which changes operation scheduling to make an efficient configuration of L0 cluster.

4.1 Power breakdown of VLIW processors

A detailed power analysis of embedded systems using VLIW processor indicates that significant amount of power is consumed in the instruction memory hierarchy. For example in Lx processor, a VLIW processor designed by Hewlett-Packard and STMicroelectronics, up to 40% of the total processor power is consumed in the instruction caches alone [8]. Figure 4.1 (a) shows the average power consumption of the VLIW processor reported in [8]. This figure shows that the instruction cache consumes a significant amount of power; 36% of total power.

An L0 buffer (a.k.a. loop buffer) is an efficient technique to reduce energy consumption in the instruction memory hierarchy [9, 10]. In most embedded applications, significant amount of execution time is spent in small program segments (which consist of loops). An L0 buffer stores these small program segments in a small buffer (SRAM or register file based) instead of a big instruction cache. Then the processor core only accesses to the buffer during the loop execution. This reduces the number of accesses to the higher level of the instruction memory hierarchy and therefore giving large energy reduction, for instance up to 60% as shown in [10].

Other components of VLIW processor, such as the data path and data memory, can also be optimized for energy efficiency. Many research communities have been devoted to the energy optimization for the components. A power management architecture eliminating the switching activity of FUs [50] reduces the energy consumed in the data path by up to 40%. The L0 buffer architecture reduces the energy of the instruction cache by up to 60%. Data memory energy

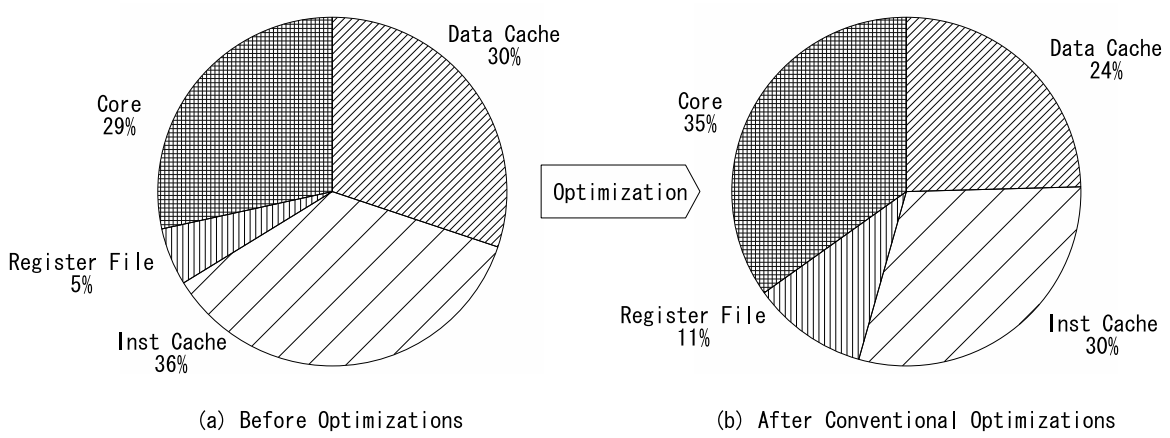


Figure 4.1: Power breakdown of VLIW processor (a) before optimizations (b) after conventional power optimizations.

can be reduced by up to 60% by code transformations at the system level [51]. With these conventional power optimizations, the power consumption can be reduced by about 50% as shown in Fig. 4.2. However, the instruction cache is still a major power bottleneck as shown in Fig. 4.1 (b).

4.2 L0 buffer in VLIW processors and L0 cluster

In a simple application of the monolithic L0 buffer to a VLIW processor, at each cycle the operations would be fetched for all the slots of the VLIW from the monolithic L0 buffer. However, such a monolithic L0 buffer is not effective as not all slots are always active. This implies that some slots would require unnecessary buffer access for NOP operation [13]. Hence, L0 cluster generation was proposed to obtain a low energy system [13, 14].

Figure 4.3 depicts an overview of a clustered VLIW processor. A VLIW instruction consists of multiple operations that are executed simultaneously. A VLIW processor consists of a number of slots and each of the different slots operates in parallel, and hence an operation can be issued from each slot every cycle. An L0 buffer provides operations to slots during loop execution, while L1 cache or higher instruction memory hierarchy directly provides operations outside of loop. Each slot is also associated with a data cluster where all the slots inside are connected to the same register file. The slots are also grouped to form an L0 cluster (instruction cluster) and these slots are associated with their respective L0 buffer.

Each L0 cluster contains associated slots, the separated L0 buffer, and an index translation controller (ITC) which controls access to the L0 buffer. In each cluster, the buffer stores only operations destined to the slots in the cluster. A loop controller (LC) gives a relative index in a loop to ITCs during loop execution, and an ITC regulates access to L0 buffer when no operation is needed to be issued. Since an ITC controls buffer access for each cluster, unnecessary accesses to the L0 buffers, i.e. fetching NOP operation, can be suppressed. Further architecture

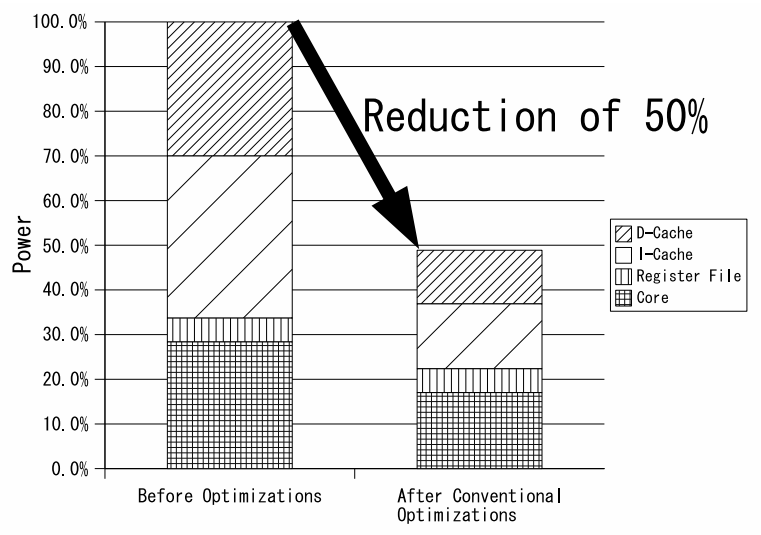


Figure 4.2: Power reduction by the conventional power optimizations.

details are provided in [13, 52].

Figure 4.4 shows an example of how to control the buffer accesses. Figure 4.4 (a) shows a schedule to be executed and a possible L0 cluster configuration is shown in Fig. 4.4 (b). In a cycle when no effective operation exists in an L0 cluster, buffer access can be regulated in the cluster, as represented in shaded boxes labeled 'INACTIVE' in Fig. 4.4 (b). Figure 4.4 (c) shows a detailed cycle by cycle execution behavior of L0 cluster 3. An ITC consists of enable flags and pointers to L0 buffer, and the LC gives a pointer to an ITC in each cluster. In the first cycle, the enable flag is true in the entry pointed by the LC, then the L0 buffer is activated and a pointer in the entry is used to fetch operations from the buffer. In the second cycle, the enable flag is false, which represents that there is no effective operations in the cycle. Therefore, a buffer access to the L0 buffer is regulated. In the next cycle, the L0 buffer is activated again and operations for the cycle are fed to slots. Note that applying L0 clusters can reduce the depth of L0 buffer itself (depth of three, while the size of loop is four), as well as the number of accesses to the buffer, and consequently contributes energy reduction significantly.

4.3 Motivation for impact of compiler

Clustering L0 buffers is effective for energy reduction. The result of L0 cluster generation is, however, sensitive to the schedule of the target application. The essence of the relevant energy model [13] is

$$E = \sum_{i=1}^{N_{clusters}} E_{LBi} \cdot Cactv_i + E_{ITCi} \cdot C, \tag{4.1}$$

where $N_{clusters}$ is the number of L0 clusters, E_{LBi} is the energy consumed for any random access, E_{ITCi} is the energy consumed by ITC for a cycle, $Cactv_i$ is the number of activated

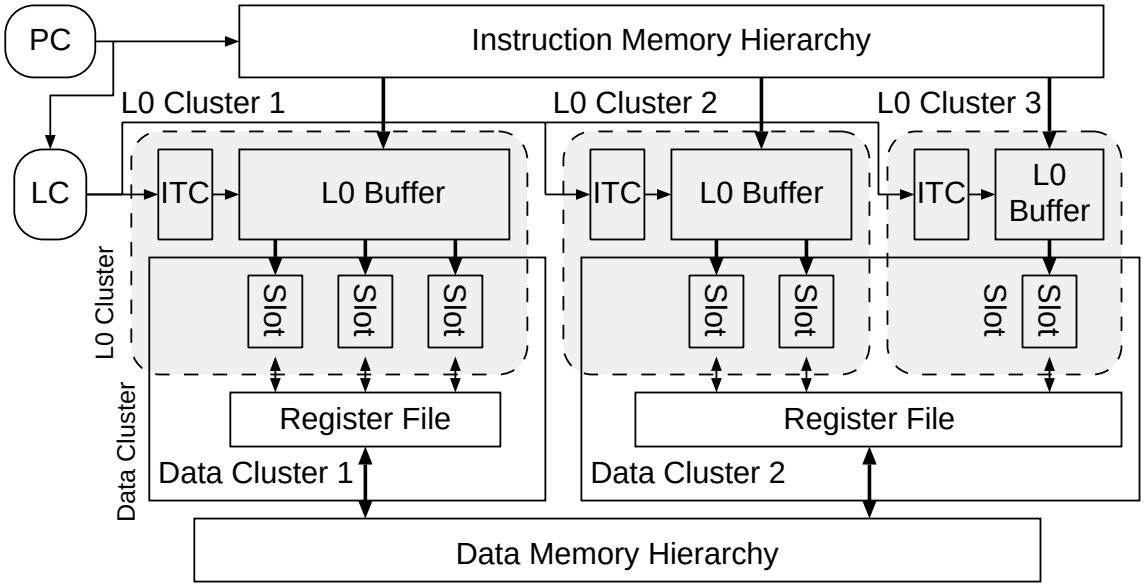


Figure 4.3: A clustered VLIW processor.

cycles in L0 cluster i, C is the number of total cycles during loop execution. E_{LBi} depends on the size of cluster.

Figure 4.5 depicts a small example to show how energy is reduced by a change of schedule[1]. An example of 5 cycle schedule in 3 slot VLIW processor is shown in Fig. 4.5 (a). One possible schedule for this is shown in Fig. 4.5 (b). In this schedule, L0 cluster 1 (L0C 1) has only one operation in cycle 5, while the cluster can contain up to two operations. If the operation is moved to another cluster, a smaller cluster (L0C 2) is activated and the larger cluster (L0C 1) can be inactivated, and consequently, it can reduce energy by 27% as shown in Fig. 4.5 (c). Therefore the basic rationale of this change in schedule from Fig. 4.5 (b) to (c) is to use smaller L0 clusters more efficiently than larger L0 clusters inefficiently.

This fact emphasizes the impact of compiler on the energy efficiency; the energy efficiency is highly depending on the initial schedule generated by a compiler even if L0 clusters are properly constructed for a given schedule.

[1]In Fig. 4.5, the buffer size and ITC size are calculated using the following equations:

$$Size_{buffer} = (Len_{op} \times \#Slots) \times Depth_{buf}, \tag{4.2}$$

$$Size_{ITC} = (Width_{flag} + Width_{index}) \times Depth_{ITC}, \tag{4.3}$$

where $Width_{flag}$ is the width of enable flag, $Width_{index}$ is $log_2(Depth_{buf})$, $Depth_{buf}$ is the same as the number of active cycles in the cluster, and $Depth_{ITC}$ is the same as the length of loop. Note that the above equations are just a simplified model than the energy model used in the experiments, which is based on Wattch power model [53].

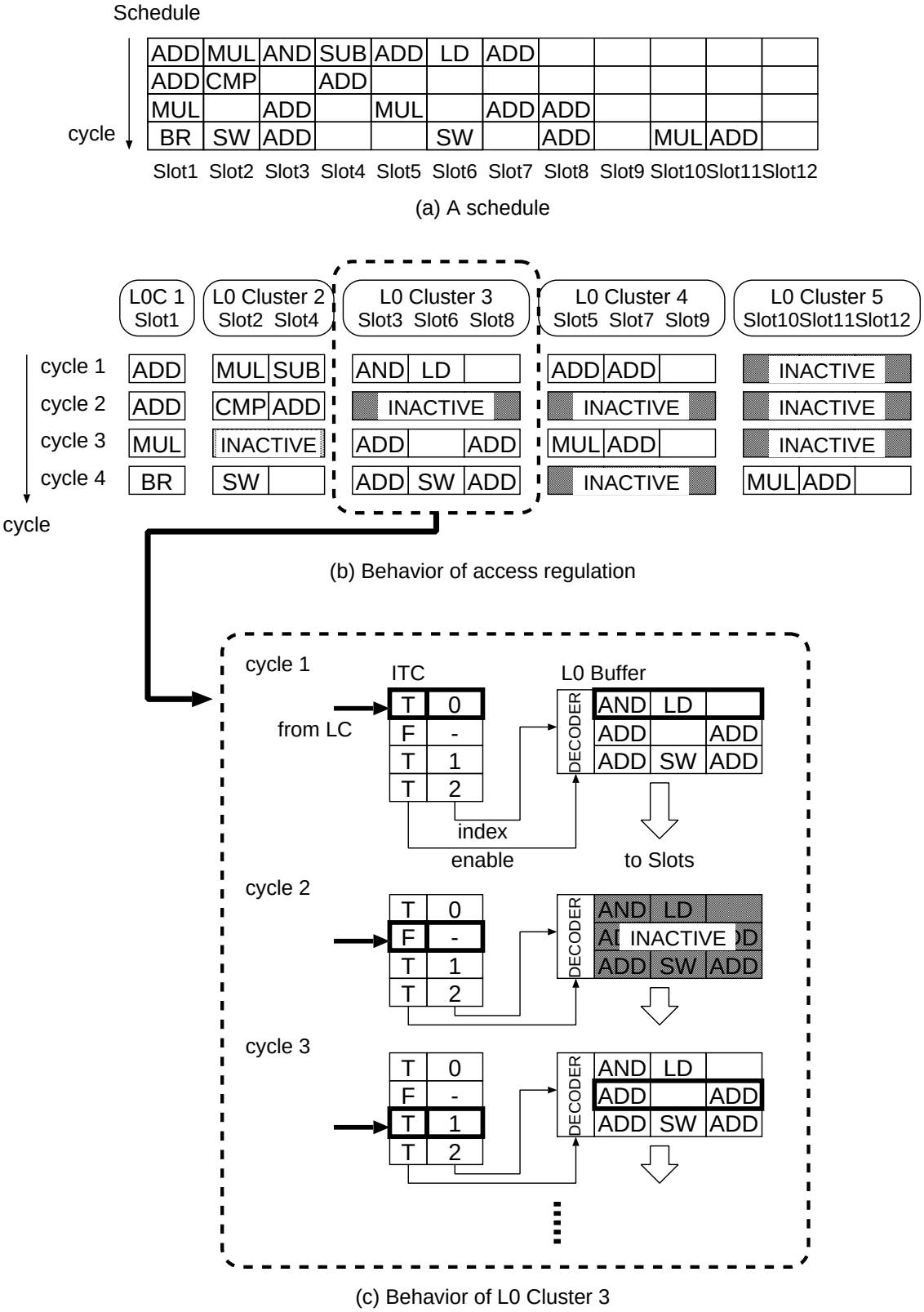


Figure 4.4: Example of regulation of L0 buffer access.

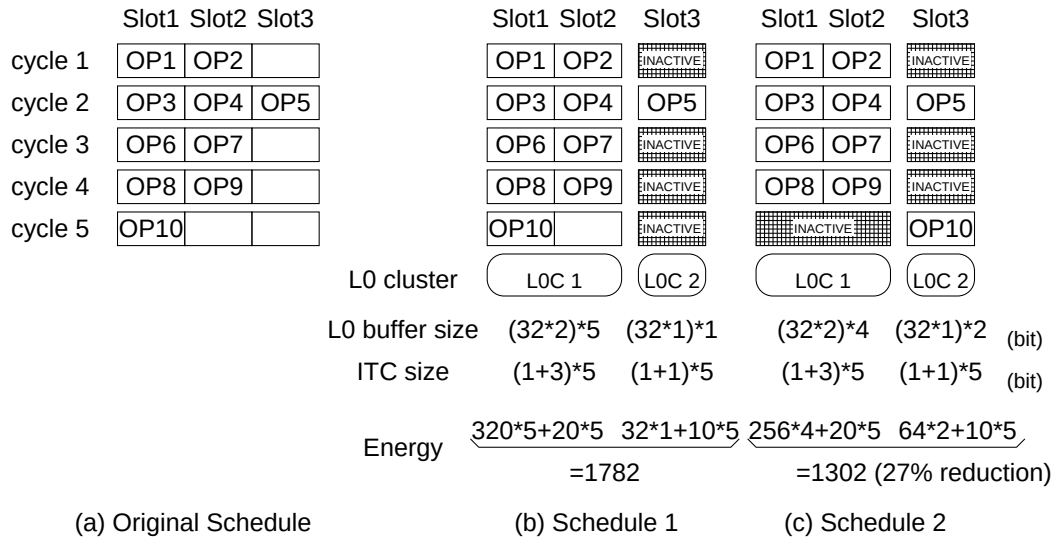


Figure 4.5: Example illustrating energy reduction by schedule change. (operation length is 32 bit)

4.4 Proposed operation shuffling algorithm on heterogeneous architectures

Figure 4.6 shows an overview of the proposed operation shuffling algorithm. Figure 4.6 (a) shows a conventional method proposed in [13]. An *initial schedule* is first obtained by compilation of the *application* on the specified *architecture* of target processor by using *retargetable VLIW C compiler*. The CRISP framework [54] is used, which is an extended version of Trimaran framework [55]. The *initial schedule* is then further analyzed by a *schedule analyzer* which generates *activation information* of slot. Finally, an *L0 cluster optimizer* finds the most energy efficient L0 cluster configuration for the information, and reports an *optimized cluster configuration* and *estimated energy* corresponding it, as output of post compilation phase. Figure 4.6 (b) shows the extended method proposed in this article. Here the *schedule analyzer* is extended to generate all possible operation-shuffled schedules (i.e. *slot activation info*), according to the *architecture information* (target processor). The output of the post compilation phase is sets of *L0 cluster configuration* and *estimated energy information* for each generated schedule. Therefore, the best schedule and optimized L0 cluster configuration for it can be obtained. For estimating the energy consumption Wattch power model [53] is used and the $0.18\mu m$ technology is assumed.

Figure 4.7 depicts a procedure in the extended schedule analyzer to shuffle operations in a cycle under constraints of slot capability. For each cycle, assignment candidates are generated by shuffling operations scheduled to the cycle (the dependencies of the operations need not be taken into account as cycle boundaries are not crossed). An assignment list includes a list of operation assignment candidates for each cycle. The list of candidates is generated from an

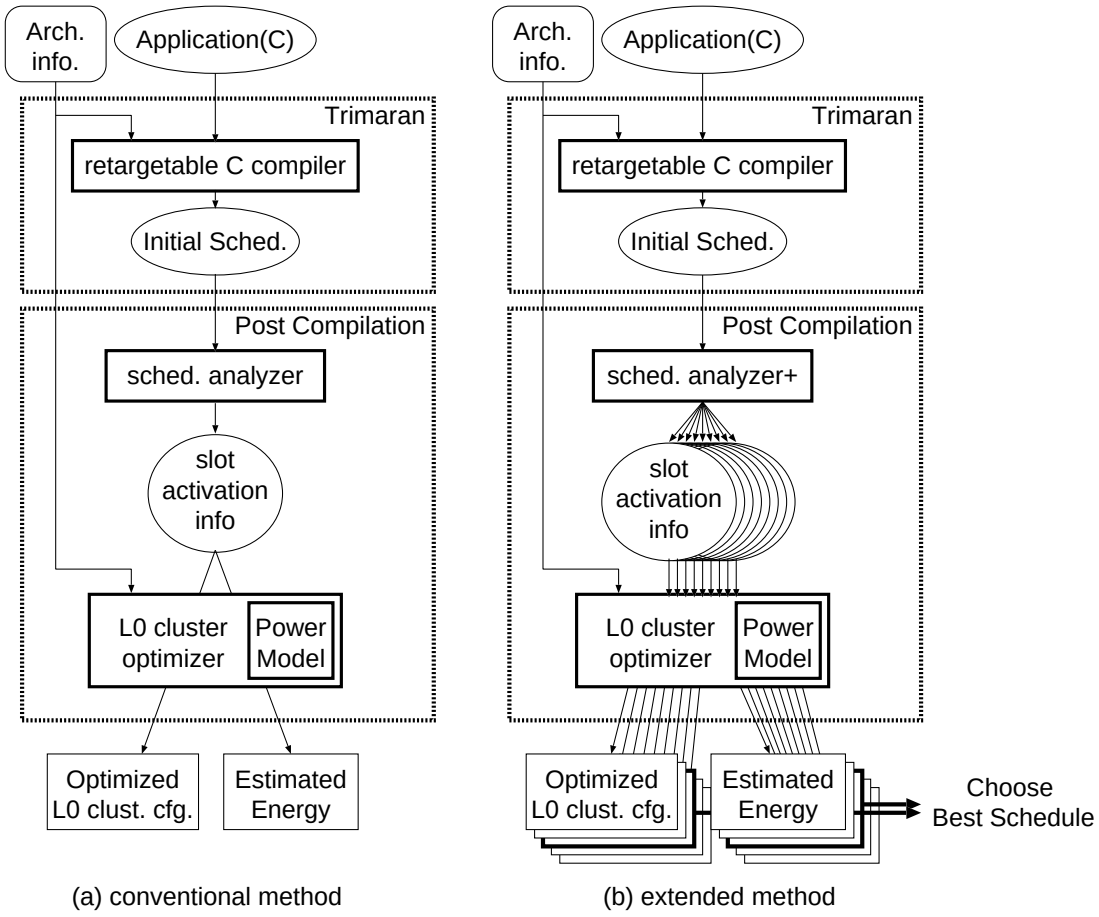


Figure 4.6: Overview of an L0 cluster configuration improvement phase (a) in the conventional way, (b) with operation shuffling (proposed method).

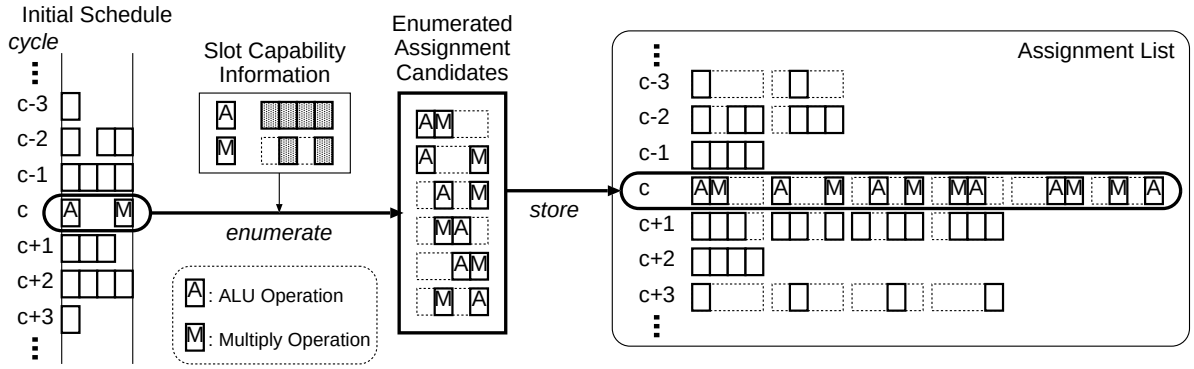


Figure 4.7: Operation shuffling in each cycle.

initial schedule, which is generated by the compiler, with slot capability information, which is extracted from architecture information given as input. Note that this operation shuffling algorithm is applied for each data cluster in a data-clustered architecture, since an operation cannot be simply moved across the border of data cluster without an inter-cluster copy operation. As the proposed shuffling algorithm is applied as a post compilation phase, new inter-cluster copy operations cannot be inserted. Section 4.6 describes details of this topic. In Fig. 4.7, an ALU operation and a multiply operation are bound in cycle c of initial schedule which is generated by the retargetable compiler. According to slot capability information (i.e. multiply operation can be issued from the second and the fourth slot, while ALU operation can be issued from all slots) which is given as the architecture information, six assignment candidates are enumerated for the ALU operation and the multiply operation in cycle c. These are candidates for schedules to be generated in the next step and stored into an entry of cycle c in the assignment list. For each cycle, operation assignment candidates are enumerated and stored into the assignment list in this way. The dependency information of the operations is not needed since the procedure does not move operations across 'cycle' boundaries. It only needs slot capability information.

From this assignment list, various schedules can be generated, as shown in Fig. 4.8. By choosing one candidate for each cycle, all possible combinations of candidates are generated. For example, by choosing the first assignment possibility for each of the cycles from $c-3$ to cycle $c+3$, schedule (1) is generated. Similarly, the second assignment possibility of cycle $c+3$ is used to generate schedule (2). This approach makes it possible to generate all possible schedules in which operations are shuffled within a VLIW instruction.

4.5 Heuristics to limit the exploration space

The approach described in Section 4.4 can generate all possible shuffled schedules, however, the exploration space becomes too huge to solve it in a realistic time. In case of applying the shuffling to an application that has $M \times S$ cycles (where M is the number of basic blocks and S is the number of cycles per basic block) and each cycle has about N candidates on average, the algorithm generates almost $N^{M \times S}$ schedules. Since $M \times S$ becomes larger than thousands in real applications, this approach is not realistic as it stands. For instance, if each basic block has $S = 20$ cycles[2], each cycle has $N = 10$ patterns of assignment candidates on average, and there are $M = 500$ basic blocks in the entire application, the size of exploration space becomes $10^{500 \times 20}$, which cannot be treated in realistic manner. Therefore, it is not practical to perform a full search based operation shuffling on the entire application.

Before introducing a heuristic for this problem, let us formulate the full search as a *global approach*. Figure 4.9 (a) illustrates the global approach, where all cycles in all the basic blocks are shuffled at a time. Here, assignment candidates are enumerated for all cycles ($M \times S$) of

[2]In the context of low-power wireless and multimedia systems, many loop transformations are applied, for instance loop transformations like loop fusion is applied to improve the locality of data and instruction accesses. When these transformations are applied the effective number of cycles are increased compared to conventional number of cycles reported for generic embedded systems (e.g. as reported in [56]). In this context the number of cycles assumed for the illustration is fairly realistic.

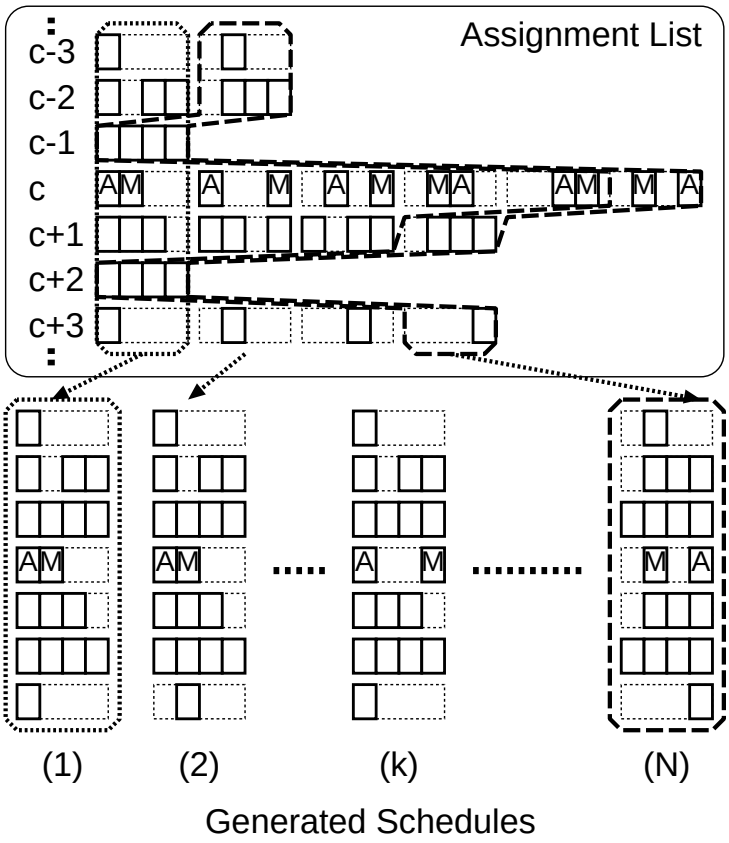


Figure 4.8: Generation of operation shuffled schedules.

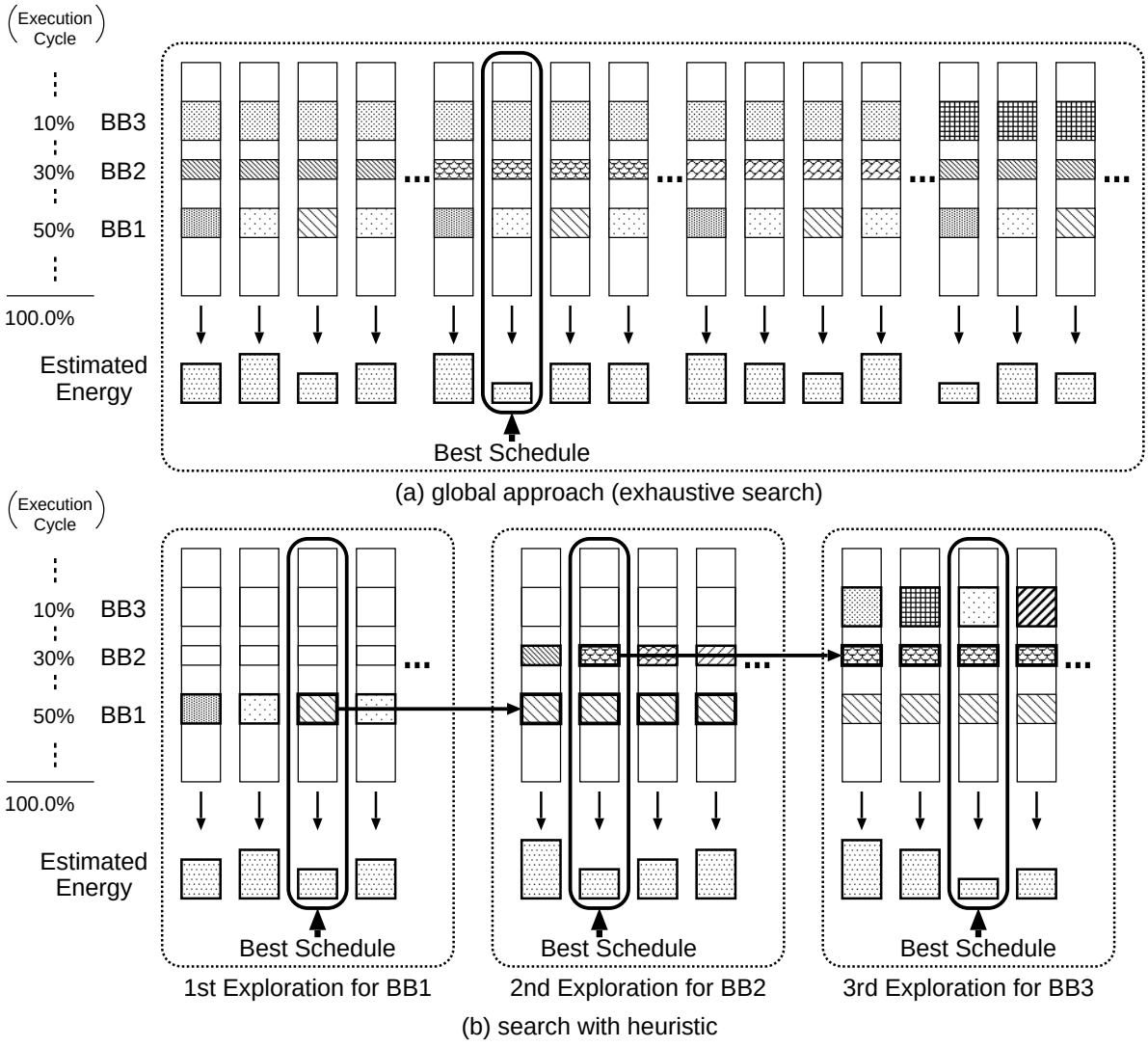


Figure 4.9: A heuristic for multiple basic blocks.

all basic blocks (N) in the target application. Further, each generated schedule is a collection of combinations of assignment candidates from each cycle. Then, all possible schedules are generated. From this list of possible schedules, the schedule (Best Schedule) which has the lowest L0 cluster energy is chosen. The L0 cluster energy is estimated as shown in the bottom part of Fig. 4.6 (a) or 4.6 (b). In the following subsections, some effective heuristics to reduce the exploration space are proposed.

4.5.1 Heuristic to shuffle one basic block at a time

As a first heuristic to reduce the complexity of the search space, the shuffling algorithm is applied to one basic block at a time instead of all the basic blocks together in the global approach. This heuristic is illustrated in Fig. 4.9 (b). Intuitively, by applying this heuristic the total schedules generated can be reduced to $M \times N^S$ from $N^{M \times S}$, which gives a linear reduction with respect to number of basic blocks (M). This heuristic is inspired by the method in [42], where a similar approach is applied to which limits the size of exploration space by considering interaction among instructions only in a basic block.

In order to apply this heuristic, the basic blocks are ordered based on a certain priority. Here, the *number of execution cycles* consumed by a basic block is utilized as an indicator for this priority. First, the algorithm ranks basic blocks according to the number of execution cycles consumed by each basic block. By this ranking, it can distinguish the most significant basic block for energy reduction; operation shuffling on this basic block is more effective rather than any other basic block. Once the operation shuffling yields the best schedule for the most significant basic block, the next operation shuffling is performed on the next cycle-consuming basic block, taking into account the shuffling result of previous first basic block. Figure 4.9 (b) shows an overview of this heuristic. First, the most significant basic block on execution cycles, BB1, is selected to be performed operation shuffling. For the second most significant basic block, BB2, operations are shuffled taking into account the shuffling result of BB1. This heuristic possibly misses a better schedule that can be obtained in the global approach, however, this would be a realistic and reasonable way to treat the entire application effectively.

4.5.2 Heuristic to limit the number of basic blocks

In addition to the above heuristic, another useful heuristic is to *limit the total number of basic blocks that are shuffled*. By applying this heuristic, the M can be reduced to $m(\leq M)$, thus the total search space will be reduced to about $m \times N^S$. In multimedia applications, 66% of total execution cycles is spent in loops of size 256 instructions or less, 51% of the total execution cycles is spent in loops of size 32 instruction cycles or less, while the size of application is a thousand to fifty thousand instructions [56]. For instance also reported else where in the literature, about 90% of execution cycles is consumed in five most frequently used loops in multimedia applications [57]. Therefore by focusing on few key basic blocks (or loop), the exploration space becomes much smaller. Note also that by focusing on most important loops the achievable energy efficiency is not severely compromised, since most time consuming loops are also the ones which consume energy.

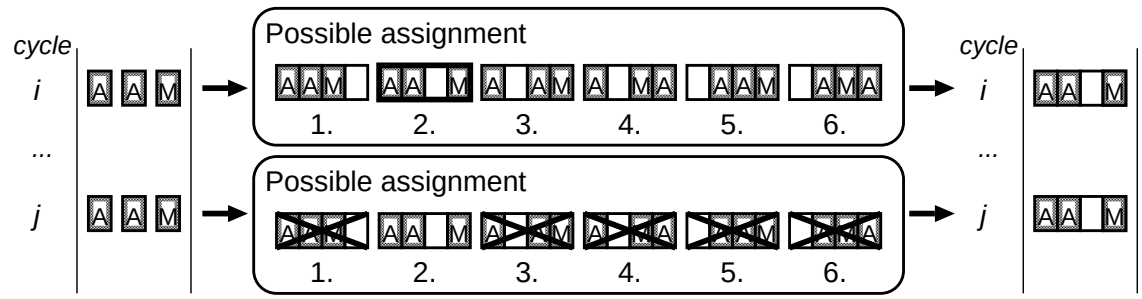


Figure 4.10: Skipping same combination heuristic.

4.5.3 Heuristics to select the combination of assignment candidates

Another set of heuristics is inspired based on the insights of operation of L0 clusters or loop buffers [13]. Specifically, these heuristics are implemented under the hypothesis that it is better that *all the active slots should be concentrated together*. Intuitively, this helps in generating smaller widths for frequently used clusters. Since smaller clusters are better for energy efficiency, this heuristic helps in achieving higher energy efficiency. However, as a side-effect this policy might lead to increase in depth of buffer of the frequently used L0 cluster. Therefore, these heuristics should be experimentally examined for a trade-off between depth and width of buffer. Even if a result gets a bit worse due to the heuristics, these heuristics will still be beneficial in reducing the exploration space. This is discussed further in Section 4.7.2.

More importantly, by applying these heuristics, it is potentially possible to reduce the number of patterns needed for each cycle to $n(\leq N)$, thus reducing the total exploration space to $m \times n^S$. It is clearly evident from the equation that by reducing the number of patterns from N to n we can reduce the search space drastically, since n is the base of the exponential factor. Hence, these set of heuristics are crucial and important for reducing the overall search space.

4.5.3.1 Skipping same combination

In a basic block, if current cycle has the same combination of operations as in the previous cycle, then the same operation assignment is used for the current cycle. For example in Fig. 4.10, if cycle i contains two ALU operations and one multiply operation and they are assigned into the first, second and fourth slots, respectively, for cycle j which contains the same combination as i, i.e. two ALU and one multiply, the same slots as i, i.e. the first, second, and fourth slots, are used. In exhaustive exploration, even if cycle j has the same combination of operations as cycle i, exploration is repeated again for cycle j. On the contrary, this heuristic skips exploration for cycle j by applying the same assignment as cycle i, which is expected energy efficient assignment, and consequently the exploration space is reduced.

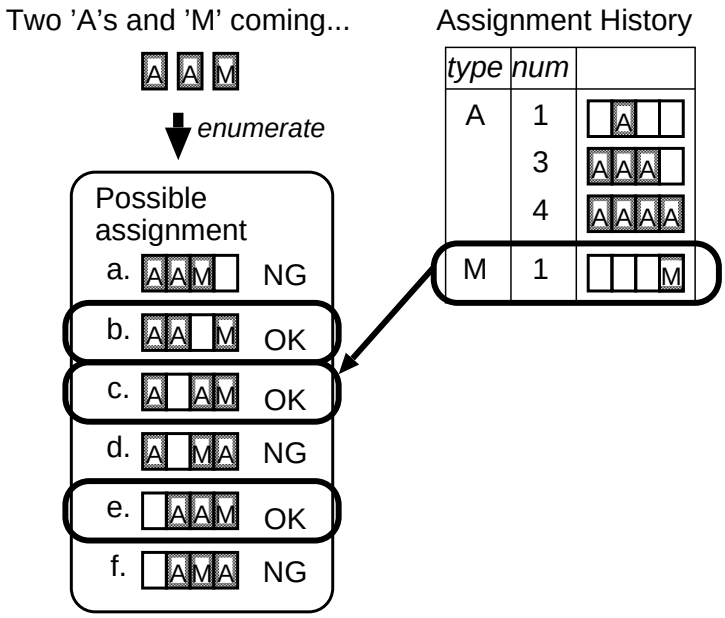


Figure 4.11: Dominance checking heuristic.

4.5.3.2 Dominance checking

In this heuristic, in a basic block, if the current cycle has the same number of operations of a certain functionality (e.g. one multiply operation) as in one of the previous cycles, then the same operation assignment as the previous cycle is used for the current cycle. Figure 4.11 explains this heuristic. For a certain cycle, two ALU operations and one multiply operation are to be scheduled. An *assignment history* is maintained to keep track of the slot assignment history for each operation type and for its number. Even if there is no exactly same combination as the previous cycles, there is an entry in the *history* which uses a multiply operation in the fourth slot. Therefore, three of the enumerated patterns (a, d, and f) which do not use a multiply operation in the fourth slot are omitted in this heuristic, and only the rest of the patterns (b, c, and e) are used for the generation of schedules. The policy of this heuristic is the same as the previous heuristic. If a certain operation is examined and a decision to assign a slot is made in the previous cycle, the same decision is applied without any further exploration in the later cycle.

4.5.3.3 Advanced dominance checking

This is an improved version of the dominance checking heuristic. Here an infeasible candidate is skipped not only in the case that the number of operations is exactly same as in one of the previous cycles for a certain functionality but also in a case that the assignment order is different. Figure 4.12 shows an example of this heuristic. In Fig. 4.12, though there are four possible assignments for three ALU operations and there is no exact match with the history,

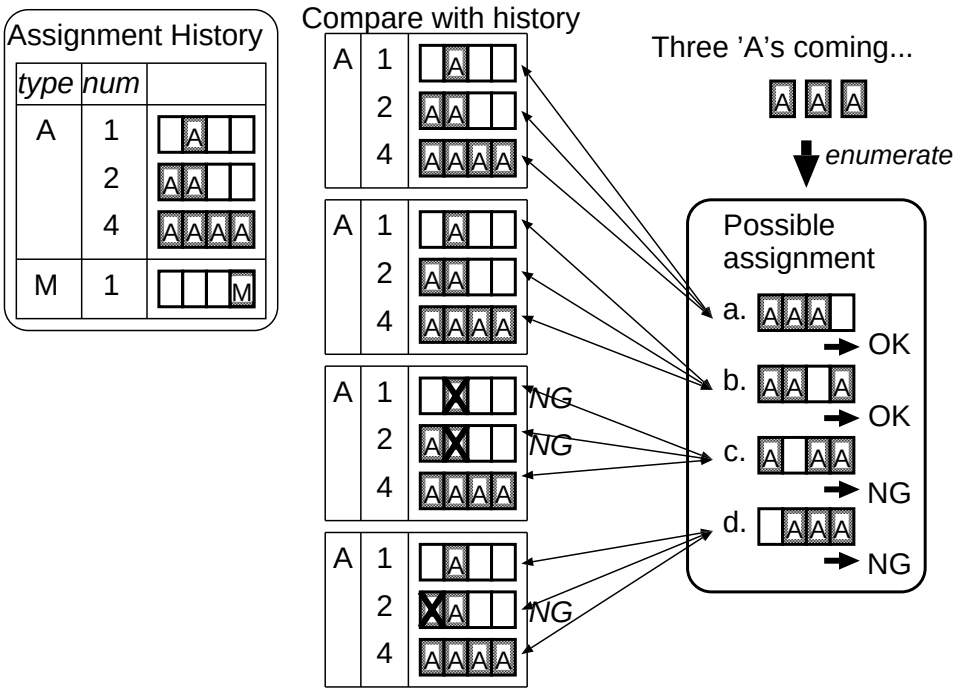


Figure 4.12: Advanced dominance checking heuristic.

```
procedure CheckPattern(candidate, cycle)
  skip ← false
  foreach type ∈ OpeTypes do
    cur ← Extract(candidate, cycle, type)
    count ← bit_count(cur)
    // Dominance checking
    prev ← History(type, count)
    if prev ≠ null and prev ≠ cur then
      skip ← true
    end if
    // Advanced dominance checking
    for n ← count + 1 to Slot_max do
      prev ← History(type, n)
      if prev ≠ null and (prev&cur) ≠ cur then
        skip ← true
      end if
    end for
    for n ← count − 1 downto 1 do
      prev ← History(type, n)
      if prev ≠ null and (prev&cur) ≠ prev then
        skip ← true
      end if
    end for
  end foreach
  if skip = false then
    if cycle is the final cycle then
      AcceptCandidate(candidate)
    else
      AddHistory(type, count, cur)
      CheckPattern(candidate, cycle + 1)
      RemoveHistory(type, count, cur)
    end if
  end if
```

Figure 4.13: Dominance and advanced dominance checking.

assignment *c* and *d* are not appropriate because they are inconsistent with the history; assignment of three ALU operations has to dominate assignment of one and two ALU operations and has to be dominated by assignment of four ALU operations. Dominating means that slots to which a fewer operations are assigned should be included in slots to which more operations are assigned. In this example of Fig. 4.12, three ALU operations have to be issued from a set of slots which are used for four ALU operations (this constraint is satisfied in all candidates in this example), and a set of slots used for one or two ALU operations has to be subset of a set of slots used for three ALU operations. In the mathematical notation, the slot assignment has to satisfy the constraint

$$i < j \rightarrow S_i \subset S_j, \tag{4.4}$$

where i, j is the number of operations and S_k is a set of slots used for k operations.

4.5.3.4 Algorithm for dominance and advanced dominance checking

Figure 4.13 shows the dominance and the advanced dominance checking algorithm. A generated $candidate$, which is a combination of assignment for each cycle, is checked whether conditions of the heuristics are fulfilled for all operation types in each $cycle$. $OpeTypes$ is a set of operation types that a VLIW processor supports, and $Slot_{max}$ is the number of slots in the processor. $Extract(cand, cyc, t)$ extracts an assignment in cycle cyc from candidate $cand$ and returns slots to which operation type t is assigned in the assignment. cur is a bit string in which each bit corresponds with a slot; a bit is true when corresponding slot is used in the cycle. $count$ is the number of true bits in cur, which is one of keys of the history table. $prev$ is a bit string representing slots used in the previous cycles. By comparing cur and $prev$, the dominance checking is done. By bitwise AND of the two bit strings, the advanced dominance checking is done. If all conditions are fulfilled, the next cycle of the candidates is checked. If it is the final cycle, $candidate$ is accepted as a result of this algorithm by $AcceptCandidate()$.

4.6 Operation shuffling for multiple data clusters

So far in the approach of operation shuffling, the assumption was that all the L0 clusters were within one data cluster. This section describes an overview of how to apply this approach when the architecture supports multiple data clusters. Since a wide VLIW processor leads to drastic increase of complexity of interconnect and size of the data register file, some VLIW processors have clustered register files. In data clustered VLIW processors, smaller register files significantly benefit energy reduction [58], though an inter-cluster copy operation is required to be added to transfer data across the border of data clusters. Since such a data clustered VLIW processor is commonly used, this section also describes a operation shuffling approach to treat VLIW processors with multiple data clusters.

In order to combine both data and L0 clusters, this thesis proposes the following approach: First the data clustering (partitioned data register files and cluster copy operation insertion) is applied and then within these constraints, L0 clustering and operation shuffling is applied. Such a phase ordering is both energy efficient and also scalable to large architectures like [39] [40]. Qualitatively, we can see that the register file energy consumption is more than the energy consumption of the L0 clusters, hence by focusing on the higher energy bottleneck first and then

applying the L0 clustering will lead to better overall efficiency. Moreover, this approach avoids any side-effects that can arise by L0 operation shuffling which could potentially affect the data clustering phase. As a consequence of this approach, in the operation shuffling approach presented in this thesis, the operations will only be shuffled within the boundary of a data cluster. Which implies that the data clustering constraints are violated, if we shuffle the operation and reassign it another L0 cluster which resides in another data cluster. Another consequence, is that we need to keep the size of an L0 cluster to be smaller than or equal to the size of a data cluster. Therefore, this thesis assumes only L0 clusters which reside in a data cluster.

Now, in order to optimize a schedule for data clustered VLIW processors, the operation shuffling described in the previous sections is performed on each data cluster separately. Under this policy, no inter-cluster copy operation is needed to be added since operations are moved within the data clustering constraints. Hence, there is no performance loss nor any interference with the data clustering phase. This idea is relatively straightforward, but it has still important meaning because most VLIW processors have multiple data clusters.

4.7 Experimental results

This section describes experimental results, which show that the proposed operation shuffling algorithm works properly and the heuristics make computational complexity much smaller.

The energy that this thesis focuses on is the energy consumed in the L0 buffers and ITCs. Since they consist of register file like storage, Wattch power model[53] has been used and 300MHz clock frequency in an $0.18\mu m$ technology node is assumed. Energy estimation is performed using the equation shown below [13]:

$$E_{L0} = \sum\{E_{LBi} \cdot Cactv_i + E_{ITCi} \cdot C\}, \tag{4.5}$$

where E_{LBi} and E_{ITCi} are energy per access of L0 buffer and ITC of L0 cluster i, respectively. $Cactv_i$ is the number of accesses to L0 buffer of the cluster i, and C is the number of total execution cycles during loop execution. E_{LBi} and E_{ITCi} are estimated from the width and depth of L0 buffer and ITC by using *Wattch* power model, and $Cactv_i$ and C are obtained by instruction level simulation in CRISP framework.

Note that, this thesis refers energy reduction as the *difference* between energy consumed in optimized L0 clusters for an initial schedule and the minimum energy consumed in optimized L0 clusters for operation shuffled schedules. In the experiments, five kinds of VLIW processor targets are used:

1. 4 slot heterogeneous single data cluster
2. 8 slot heterogeneous single data cluster
3. 10 slot heterogeneous single data cluster
4. 8 slot homogeneous single data cluster
5. 2 data clusters, with 5 heterogeneous slots per data cluster.

Table 4.1: Slot capability of 8 slot heterogeneous VLIW processor.

Slot	0	1	2	3	4	5	6	7
alu	*	*	*	*	*	*	*	*
shift			*				*	
mult/div	*				*			
fp alu			*	*			*	*
fp mult/div	*				*			
load/store	*	*	*	*	*	*	*	*
branch	*	*	*	*	*	*	*	*

Table 4.2: Slot capability of 10 slot heterogeneous VLIW processor.

Slot	0	1	2	3	4	5	6	7	8	9
alu			*	*	*			*	*	*
shift			*	*	*			*	*	*
mult/div	*					*				
fp alu			*	*	*			*	*	*
fp mult/div	*					*				
load/store		*					*			
branch			*	*	*			*	*	*

Table 4.3: Slot capability of 4 slot heterogeneous VLIW processor.

Slot	0	1	2	3
alu	*	*	*	*
shift			*	
mult/div	*			
fp alu			*	*
fp mult/div	*			
load/store		*		
branch				*

Table 4.4: Slot capability of 2 data cluster 5-5 slot heterogeneous VLIW processor.

Slot	0	1	2	3	4	5	6	7	8	9
Data cluster	0					1				
cluster copy	*					*				
alu		*	*	*	*		*	*	*	*
shift				*					*	
mult/div		*					*			
fp alu				*	*				*	*
fp mult/div		*					*			
load/store			*					*		
branch	*	*	*	*	*	*	*	*	*	*

Table 4.1, Table 4.2, Table 4.3, and Table 4.4 show slot capability of 8, 10, 4, and 5-5 slot heterogeneous VLIW processors, respectively. For instance, the 8 slot heterogeneous VLIW processor can issue an load/store operation from all slots, while the 10 slot heterogeneous VLIW processor can issue from only the second and seventh slots.

Note that this thesis introduces three approaches:

1. global approach (full search without any heuristics)
2. exhaustive exploration (global approach + Section 4.5.1 and Section 4.5.2)
3. exploration with heuristics (exhaustive exploration + Section 4.5.3).

In this experiments, only *exhaustive exploration* and *exploration with heuristics* are compared. It is believed that it is straight forward to evaluate the benefits of heuristics to shuffle one basic block at a time (Section 4.5.1) and heuristics to limit the number of basic blocks (Section 4.5.2). These are reasonable heuristics, as many researchers have employed these in the past [42]; [56].

4.7.1 Potential gain of operation shuffling

First, the feasibility of the operation shuffling methodology and its potential gain of energy reduction are examined. Table 4.5 shows energy reduction by operation shuffling on the

Table 4.5: Energy reduction for MPEG2 encoder on 8-slot Hetero VLIW.

Initial Energy (mJ)	25.22
Maximum Energy (mJ)	26.53
Minimum Energy (mJ)	19.11
Reduction (Initial-Minimum)	24.2%

Table 4.6: Operation shuffling on multiple BBs (MPEG2 Encoder@8 slot Hetero).

# Shuff. BBs	Energy (mJ)	Energy Red.	Shuff. Cycles
(Initial)	25.22	–	0.0%
1	19.11	24.2%	46.7%
2	19.09	24.3%	50.4%
3	18.86	25.2%	54.1%
4	18.84	25.3%	56.5%
5	18.81	25.4%	59.0%

most cycle-consuming basic block for MPEG2 encoder benchmark on an 8-slot heterogeneous VLIW processor. By applying the exhaustive exploration[3] (exploring all the alternatives) about 29400 schedules are generated, which have various values of energy. Among these alternatives, the best candidate is picked which achieves maximum energy reduction (refer to Fig. 4.6 (b)). The best candidate, in this case, is a modified schedule whose energy consumption is reduced by up to 24.2% compared with the initial schedule. The best L0 cluster configuration in this case is: two clusters of two slots and six slots, which have L0 buffers whose depth of 110 and 25 instructions, respectively.

Next, operation shuffling is performed not only on the most significant basic block but also on the other significant basic blocks (based on heuristics described in Section 4.5.2). Table 4.6 shows results of operation shuffling on multiple basic blocks for MPEG2 encoder on the 8-slot VLIW processor. The first and second columns show the number of shuffled basic blocks and estimated energy, respectively. The third column indicates energy reduction compared to the initial energy. The fourth column represents how many execution cycles are shuffled, i.e. how many execution cycles the shuffled basic blocks contribute to. These results indicate that energy reduction is going up when the number of applied basic blocks is increasing. It becomes saturated after a few basic blocks are applied, which consume 50% to 60% of the total execution time.

The potential gain of energy reduction with L0 cluster generation is up to 63%, as shown in [13]. This experimental results expose the gain can be improved furthermore up to 25.4% by the proposed operation shuffling.

4.7.2 Quality of pruning heuristics

This section evaluates the quality of the heuristics (Section 4.5.3) both in terms of the achievable energy efficiency and also in terms of reduction in the search space.

Table 4.7 shows the results of exhaustive exploration and exploration with the heuristics for

[3]The *global approach* has not been compared with a scheme which shuffles for each basic block, since we can imagine how huge the exploration space of the *global approach* would become. And also, even after introducing the heuristic to limit shuffled basic blocks (Section 4.5.2), the exploration space is still big, as shown in the exploration space of the exhaustive exploration in Table 4.7. Therefore, the proposed heuristics introduced in Section 4.5.3 are still required.

Table 4.7: Minimum energy comparison between exhaustive exploration and with heuristics (Single BB).

Architecture	Benchmark	Exhaustive Expl.		with Heuristics		BB
		Expl. Space	Energy Red.	Expl. Space (Red.)	Energy Red. (Deg.)	size
8 slot Hetero	MPEG2 encoder	117600	17.8%	12600 (89.3%)	17.8% (0.0%)	2
8 slot Hetero	gsm encoder	117600	11.7%	8400 (92.9%)	11.6% (0.1%)	2
8 slot Hetero	adpcm decoder	1835008	1.0%	168 (99.9%)	0.0% (1.0%)	7
8 slot Homo	g721 decoder	114688	2.9%	224 (99.8%)	2.8% (0.1%)	6
8 slot Homo	g721 encoder	229376	5.0%	56 (100.0%)	5.0% (0.0%)	5
10 slot Hetero	g721 decoder	6480	1.7%	60 (99.1%)	1.2% (0.5%)	6
10 slot Hetero	g721 encoder	15552	2.2%	12 (99.9%)	0.0% (2.2%)	5
10 slot Hetero	gsm encoder	1800	10.5%	360 (80.0%)	10.5% (0.0%)	2
5-5 slot Hetero	adpcm decoder	960	0.8%	30 (96.9%)	0.8% (0.0%)	7
5-5 slot Hetero	g721 decoder	480	2.3%	60 (87.5%)	0.5% (1.8%)	6
5-5 slot Hetero	sha	9216	1.2%	24 (99.7%)	0.0% (1.2%)	12

some benchmarks and architectures. The third and fifth columns show the size of exploration space (i.e. generated schedules) of exhaustive exploration[4] and exploration with heuristics, respectively. In the fifth column, the reduction of exploration space is also shown. The fourth and sixth columns indicate the maximum energy reduction over an initial schedule among schedules generated by exhaustive exploration and exploration with heuristics, respectively. The sixth column also shows the degradation of energy efficiency. The seventh column shows the size of basic block that is shuffled. In this experiment, all of the proposed heuristics are applied (Sections 4.5.1, 4.5.2 and 4.5.3) and the operation shuffling is applied to the most significant basic block. In summary, Table 4.7 shows that the exploration space reduction is around 90%, and energy reduction values of exhaustive exploration and with heuristics are almost the same; degradation of less than 1% on average. From this table, it is clear that the proposed heuristics obtain near optimal solution with significantly reduced exploration space.

As described in Section 4.5.3, the proposed heuristics are developed under the hypothesis that it is better to make a frequently used cluster smaller. In Table 4.7, exploration with the heuristics generates a little worse results in some cases. This is because the heuristics omit too much exploration space. Nevertheless, the degradation of results, i.e. difference between the energy reductions, is still small and reduction of exploration time is much more valuable: we can optimize more basic blocks in the saved time.

In order to confirm that the difference in energy reduction between exhaustive approach and heuristic approach is noticeable small, the operation shuffling is applied not only to the most significant basic block but also to the following significant basic blocks as indicated already with the heuristics of Section 4.5.2. Figure 4.14 shows a comparison of exhaustive exploration and exploration with the heuristics for multiple basic blocks. In Fig. 4.14, two lines represent energy reduction of the two exploration ways and bars represent accumulated exploration space, i.e. the total number of generated patterns. Because of the explosion of the exploration space, operation shuffling is performed only on 30 basic blocks in the exhaustive approach. The figure shows that the proposed heuristics can reduce exploration space by more than 90% without

[4]the exhaustive exploration in Table 4.7 also does shuffling for each basic block (not the *global approach*).

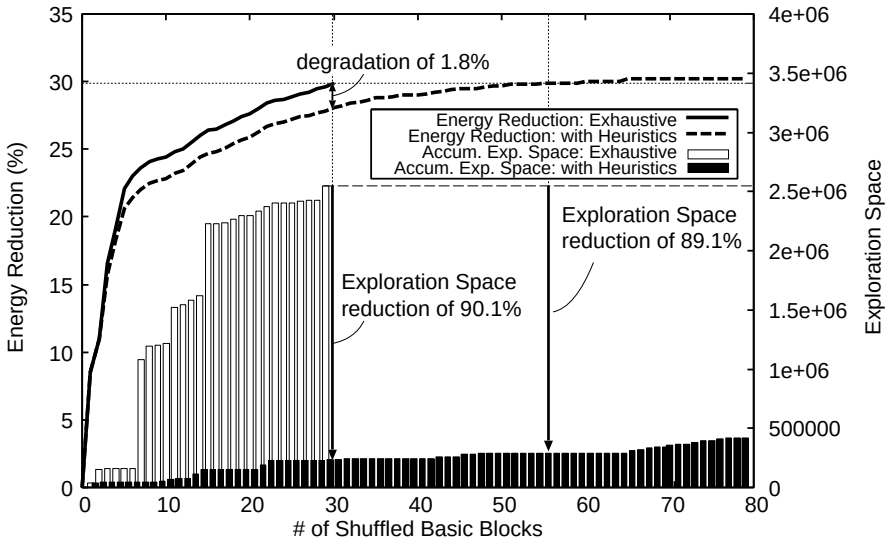


Figure 4.14: Efficiency of the heuristics (epic@8 slot Homo).

significant degradation of quality of results, only 1.8% (28.1% over 29.9%), after shuffling of 30 basic blocks. The proposed heuristics reduce more than 90% of exploration space, while the energy reduction is almost the same. This figure also shows that the heuristics can achieve better result of 30.2% after shuffling 79 basic blocks, with still more than 80% less computational time than exhaustive exploration of 30 basic blocks. The exhaustive exploration would surpass the heuristics if the same amount of basic blocks are shuffled, however, the computational effort for it is beyond the realistic limits. Though exploration with the heuristics requires operation shuffling of more basic blocks to achieve the same quality of the exhaustive exploration, the size of total exploration space is still much smaller than the exhaustive exploration.

Figure 4.15 shows a comparison of distribution in the exploration space between exhaustive exploration and with the heuristics. The x-axis represents the period of estimated energy; from minimum to maximum estimated energy of exhaustive exploration is divided into ten periods (period 1 to period 10). Generated schedules are counted in one of the periods, and the y-axis represents the percentage of distribution for each periods. The figure also shows distribution of random exploration, in which schedules are randomly selected from the result of exhaustive exploration but the number of selected schedules is the same as the heuristics.

In Fig. 4.15, the black bar (heuristics) almost always higher than other bars in period 1 to 4. This means that the proposed heuristics mainly generate energy efficient schedules rather than the exhaustive exploration or the random exploration. This figure implies that the proposed heuristics can efficiently omit the part of the exploration space that is less relevant as opposed to random selection approach.

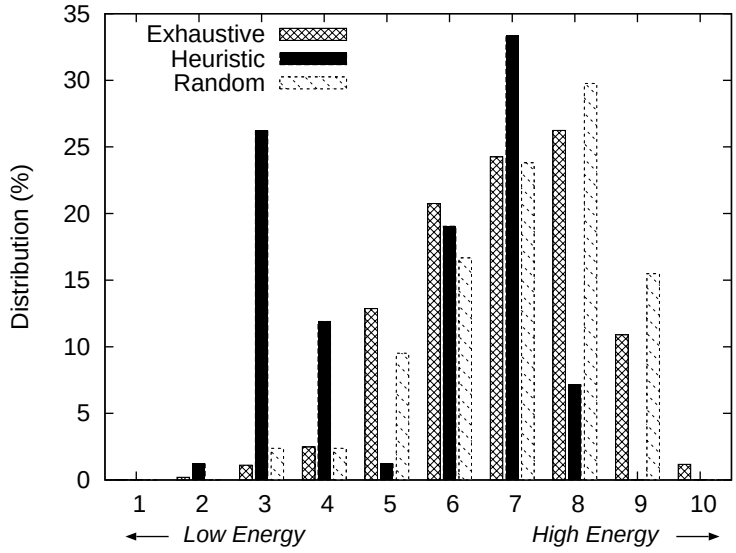


Figure 4.15: Frequency distribution of energy of generated schedules (adpcm decoder@8 slot Hetero).

4.7.3 Evaluation on multimedia benchmarks and different architecture flavors

In order to evaluate the proposed operation shuffling methodology and to examine how much potential gain realistic applications have, the methodology is applied to multimedia benchmarks [59] and for different architecture flavors as described earlier. Figure 4.16 shows comparison of energy reduction between architectures. From this figure, we can say that energy reduction is notable in a homogeneous architecture, but a heterogeneous architecture has still a good reduction.

Figure 4.17, 4.18, 4.19, 4.20, and 4.21 show the results of energy reduction for 8 slot heterogeneous, 10 slot heterogeneous, 8 slot homogeneous, 4 slot heterogeneous, and 5-5 slot heterogeneous VLIW processors, respectively.

In these figures, the x-axis represents the percentage of execution cycles which are consumed by shuffled basic blocks, and y-axis represents estimated energy of L0 buffers (smaller number implies higher efficiency). Each line for a particular benchmark indicates that, as more basic blocks are shuffled, the energy consumption decreases, since operation shuffling is applied on more basic blocks. It is to be noted that the lines for a particular benchmark do not reach 100% consumed cycles on x-axis. This is because, in a particular benchmark not all basic blocks are shuffled and only basic blocks that can be mapped on to the L0 clusters are considered [52]. For instance, basic blocks that are not in a loop body or basic blocks that are too large to store in the loop buffers are not mapped on to the L0 clusters. For example, in Fig. 4.17 for ‘MPEG2 encoder’ benchmark, about 60% of basic blocks are mapped on to the L0 clusters,

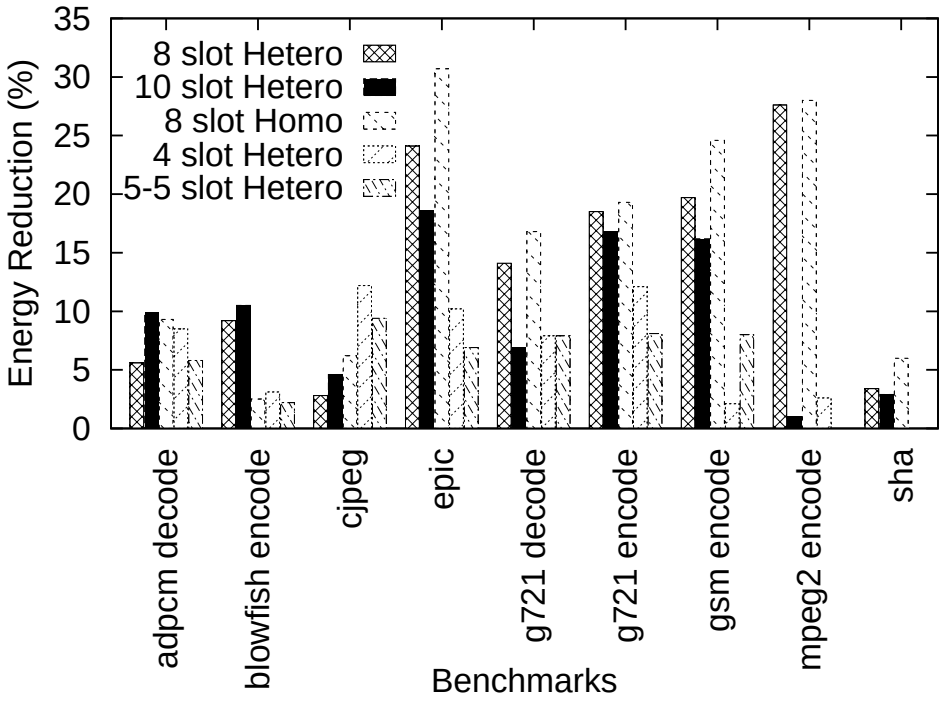


Figure 4.16: Energy reduction of all benchmarks.

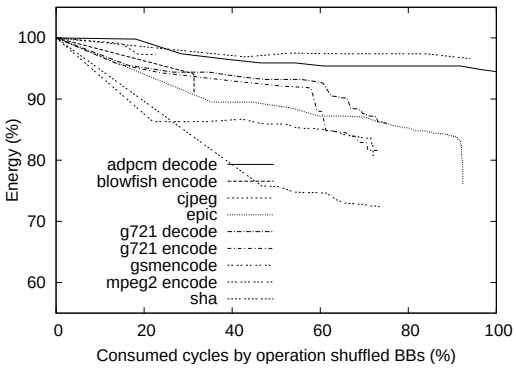


Figure 4.17: Energy reduction by shuffling operations in multiple BBs (8 slot Hetero).

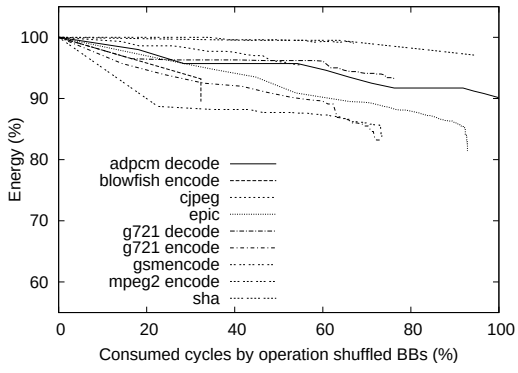


Figure 4.18: Energy reduction by shuffling operations in multiple BBs (10 slot Hetero).

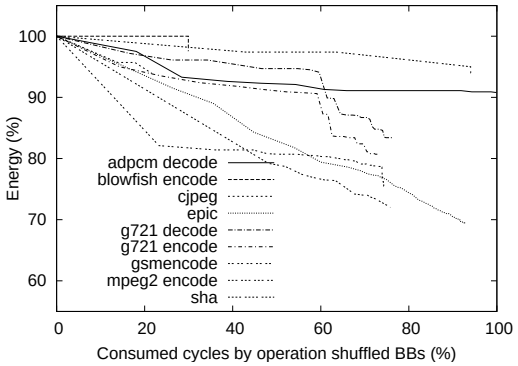


Figure 4.19: Energy reduction by shuffling operations in multiple BBs (8 slot Homo).

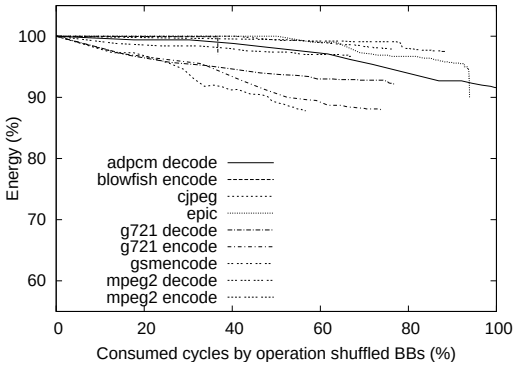


Figure 4.20: Energy reduction by shuffling operations in multiple BBs (4 slot Hetero).

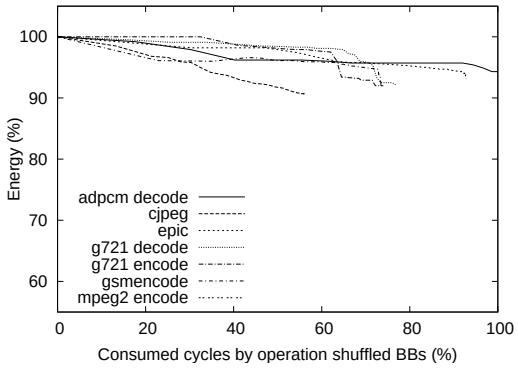


Figure 4.21: Energy reduction by shuffling operations in multiple BBs (5-5 slot Hetero).

and by shuffling these basic blocks about 20% of the energy is reduced.

It is also to be noted that the x-axis does not represent the number of operation shuffling but the percentage of shuffled execution cycles over the total execution cycles. For example of 'epic' in Fig. 4.17, the energy seems to be decreased by about 10% at the point of 92% on x-axis. This is because there are a lot of insignificant basic blocks that consumes less execution cycles in this case. By shuffling these basic blocks the energy can be decreased, however, the shuffled execution cycles does not change so much since the contribution for execution cycles is not so much in the basic blocks. Even though over 100 basic blocks are shuffled, the total execution cycles of these basic blocks is only about 1%. However, the depth of L0 buffer is managed and decreased by shuffling operations for the basic blocks; since a too deep L0 buffer causes the increase of its power, the shuffling leads to the energy reduction even if the contribution of execution cycles is not high. For example of above case, after shuffling the 100 insignificant basic blocks, the depth of the deepest L0 buffer changed from 72 to 45, and it reduced the power of the buffer by 26%.

As it can be observed from these figures, for some benchmarks, energy reduction is not notable even though most of the basic blocks are shuffled. For example, in 'adpcm decoder' in Fig. 4.17, though almost all of the execution cycles are shuffled, the energy reduction is limited to less than 6%. The reason is most probably because instruction level parallelism (ILP) in such benchmarks is low (less than 1.5 operations per cycle). For low ILP applications, the initial L0 cluster generation and the corresponding schedule is already very efficient. The available freedom to optimize L0 clusters is already used by the L0 cluster optimization procedure (refer Fig. 4.6). By further shuffling operations with in a cycle, the energy efficiency cannot be improved further, unless the shuffling algorithm is applied over the cycle boundaries. This does not imply that low ILP is a limitation of the proposed method, but that the initial schedule is already optimized by the L0 cluster optimizer. Further discussion a relation between ILP and energy reduction appears in Section 4.7.5.

On the contrary, too high ILP would not be beneficial for operation shuffling as the freedom of operation shuffling is limited if all slots are full. Then, it is limited in terms of gain by operation shuffling, however, having all the slots full is already much beneficial in terms of performance and energy; it is an ideal schedule that a compiler always pursues. The proposed operation shuffling achieve a further reduction on the energy that a compiler missed optimizing.

Sometimes energy reduction gets worse when a certain basic block is shuffled. This is because the heuristics omit the best schedule candidate. However, the degradation is still small compared with overall energy reduction. Therefore, the proposed operation shuffling methodology and heuristics are very beneficial for energy reduction.

As described in Section 4.6, for a data clustered architecture operation shuffling methodology is applied to a schedule per each data cluster. In Table 4.8 the entries corresponding to *5-5 slot Hetero* correspond to a clustered VLIW architecture with 2 data clusters, with 5 slots in each cluster. From the results it can be seen that by applying operation shuffling for each data cluster, relatively significant energy can be further reduced. Note that the energy reduction in this table refers to *additional* reduction over L0 cluster optimizer in Fig. 4.6.

Table 4.8: Relation between energy reduction and shuffled cycles.

Architecture	Benchmark	IPC	Energy Redct.	Shuff. Cycles
8 slot Hetero	adpcm decode	1.48	5.6%	99.9%
8 slot Hetero	blowfish encode	2.06	9.2%	31.3%
8 slot Hetero	cjpeg	1.38	2.8%	22.7%
8 slot Hetero	epic	2.02	24.1%	92.4%
8 slot Hetero	g721 decode	1.20	14.1%	75.1%
8 slot Hetero	g721 encode	1.24	18.5%	73.1%
8 slot Hetero	gsm encode	2.19	19.7%	72.2%
8 slot Hetero	MPEG2 encode	3.00	27.6%	73.5%
8 slot Hetero	sha	2.61	3.4%	94.1%
10 slot Hetero	adpcm decode	1.48	9.9%	99.9%
10 slot Hetero	blowfish encode	2.12	10.5%	32.3%
10 slot Hetero	cjpeg	1.39	4.6%	55.0%
10 slot Hetero	epic	2.03	18.6%	92.9%
10 slot Hetero	g721 decode	1.19	6.9%	76.4%
10 slot Hetero	g721 encode	1.27	16.8%	73.5%
10 slot Hetero	gsm encode	2.24	16.2%	73.5%
10 slot Hetero	MPEG2 encode	4.22	1.0%	67.0%
10 slot Hetero	sha	2.61	2.9%	94.4%
8 slot Homo	adpcm decode	1.48	9.3%	99.9%
8 slot Homo	blowfish encode	1.94	2.5%	29.9%
8 slot Homo	cjpeg	1.35	6.2%	21.7%
8 slot Homo	epic	2.03	30.7%	92.9%
8 slot Homo	g721 decode	1.18	16.8%	76.2%
8 slot Homo	g721 encode	1.26	19.3%	72.9%
8 slot Homo	gsm encode	2.30	24.6%	74.5%
8 slot Homo	MPEG2 encode	3.06	28.0%	75.8%
8 slot Homo	sha	2.61	6.0%	94.1%
4 slot Hetero	adpcm decode	1.44	8.5%	99.9%
4 slot Hetero	blowfish encode	1.97	3.1%	36.7%
4 slot Hetero	cjpeg	1.34	12.2%	56.7%
4 slot Hetero	epic	2.06	10.2%	93.9%
4 slot Hetero	g721 decode	1.17	7.9%	76.8%
4 slot Hetero	g721 encode	1.26	12.1%	73.8%
4 slot Hetero	gsm encode	1.98	2.1%	76.2%
4 slot Hetero	MPEG2 decode	1.55	3.9%	70.0%
4 slot Hetero	MPEG2 encode	2.30	2.6%	88.3%
4 slot Hetero	sha	2.15	0.0%	95.2%
5-5 slot Hetero	adpcm decode	1.44	5.8%	99.9%
5-5 slot Hetero	blowfish encode	1.97	2.2%	36.7%
5-5 slot Hetero	cjpeg	1.36	9.4%	56.2%
5-5 slot Hetero	epic	1.84	6.9%	92.8%
5-5 slot Hetero	g721 decode	1.15	7.9%	77.0%
5-5 slot Hetero	g721 encode	1.25	8.1%	74.0%
5-5 slot Hetero	gsm encode	2.48	8.0%	73.5%
5-5 slot Hetero	MPEG2 decode	1.56	5.8%	52.4%
5-5 slot Hetero	sha	2.16	0.0%	95.5%

4.7.4 Discussion on operation shuffling over cycle boundaries

The shuffling over the cycle boundaries should achieve more energy gain. A method proposed in this thesis only moves an operation within cycle boundaries. The method already yields a good result as shown in the previous sections, however, it can be improved furthermore by allowing move across the cycle boundaries. Some operations can move to another cycle unless the data dependency between related operations is violated. Hence, there would be more opportunities to optimize operation scheduling for energy efficiency.

The complexity of the shuffling over the cycle boundaries will, however, become too high since an operation has more freedom to move not only to different slots but also to different cycles. To manage this problem, another new and different algorithm (heuristic) is needed to efficiently prune the exploration space. However, the proposed heuristics, which limit the exploration space during operation shuffling within a cycle, can be utilized for exploration space reduction of shuffling over the cycle boundaries. Intuitively, it is a realistic assumption that an operation can also move to a different issue-slot when it moves to another cycle. Then, after moving to different cycles, the same technique for shuffling within a cycle can be used.

To further improve the effectiveness of shuffling over the cycle boundaries, a software multithreading approach would be beneficial, which combines multiple independent loops into one loop to improve performance [60]. Since independent loops are integrated into one loop, each operation has less dependency for each other in the loop. This gives an operation more freedom to move to different cycles and consequently more opportunity for energy reduction.

Shuffling operations over the cycle boundaries is, however, beyond the scope of this thesis and it will be addressed in the future work.

4.7.5 Relation between ILP and energy reduction

Table 4.8 shows the detailed result which contains IPC (instruction per cycle) as well as the energy reduction. This thesis refers to an average IPC over all basic blocks in an application as an *overall IPC*. The overall IPC represents a characteristic of target application well.

Figure 4.22 shows a relation between the overall IPC and the energy reduction. The x axis represents the energy reduction due to operation shuffling. The y axis represents the overall IPC multiplied by the percentage of shuffled cycles. The IPC is scaled with the shuffled cycles since the energy reduction is assumed to be small when only few cycles are shuffled even if the IPC is large. Though it seems there could be a correlation between IPC and the energy reduction, this is not the case. In Fig. 4.22, we can see just a weak correlation between IPC and the energy reduction.

There would be other metrics for IPC; Fig. 4.23 explains three versions of IPC that are referred in this thesis. A *shuffled IPC* is an average IPC over basic blocks that are shuffled. An *L0 buffered IPC* is an average IPC over basic blocks that are running on L0 buffer. The shuffled IPC also shows a characteristic of application, however, it can change depending on how many basic blocks are shuffled. Therefore, the shuffled IPC might be an extreme value if only few basic blocks can be shuffled due to the explosion of exploration space, ex. ‘MPEG2 encoder’ on 10 slot heterogeneous as discussed later. Similarly, the L0 buffered IPC can vary depending on which basic blocks are decided to be stored in L0 buffer. Since an algorithm that chooses basic blocks to be stored in L0 buffer [52] is sensitive to a configuration of target

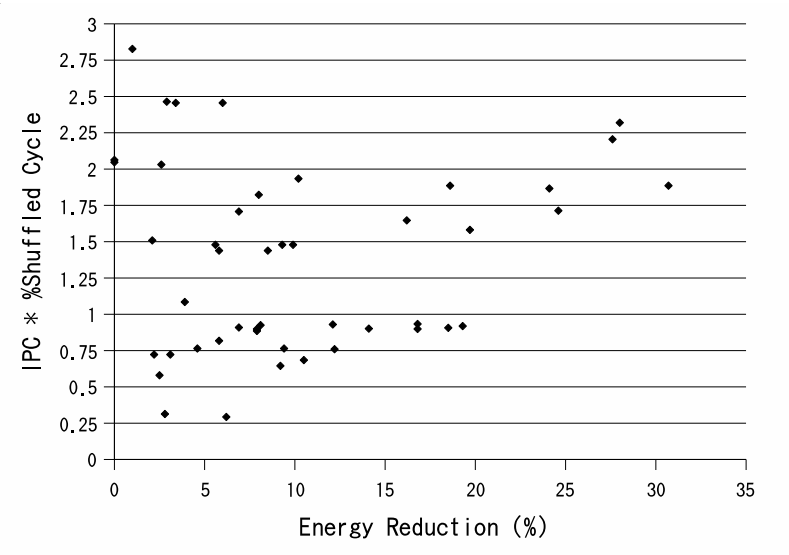


Figure 4.22: Relation between overall IPC and energy reduction due to shuffling.

VLIW processor architecture, the L0 buffered IPC is also not enough to be a representative characteristic of application. Hence, the overall IPC is used in order to examine a relation between ILP and the energy reduction. Note also that there is no stronger correlation than the overall IPC even if another IPC is employed.

One of reasons of the weak correlation is assumed that the initial energy is already good; the energy gain is not notable in such a case in spite of large IPC. For example, the energy reduction of 'sha' is not so high, 3.4% in 8 slot heterogeneous, while the overall IPC is 2.61. This is because the initial energy is already good; in the case of 'sha', a schedule generated by a compiler ('Initial Sched.' in Fig. 4.6 (b)) is very similar to a schedule that is to be generated by the proposed heuristics (the best of 'slot activation info' in Fig. 4.6 (b)). Since the energy of these two schedules are compared, the energy reduction is not notable. However, the large IPC actually yields larger variations of schedules; the energy reduction over the maximum energy is examined, rather than the energy reduction over the initial energy. Table 4.9 shows the result. As shown in the table, the energy reduction over the maximum energy is 12.3% in this case. Though it is believed that a larger IPC leads to a larger variation of energy, the energy reduction over the initial energy, which is more important than the reduction over the maximum energy from a practical viewpoint, has no direct relation to the IPC. However, this does not mean to degrade the value of the proposed method. Though there is not a strong correlation between ILP (IPC) and the energy reduction, the proposed method still achieves significant energy reduction in most of cases.

Note that 'MPEG2 encoder' on 10 slot heterogeneous is an extreme case, where the energy reduction is limited to 1.0% while the overall IPC is 4.22. This is because the most significant basic block in the case contains 9 operations and they can be scheduled in one cycle. Therefore, IPC of the basic block is 9.0, however there are not so much freedom left to shuffle operations

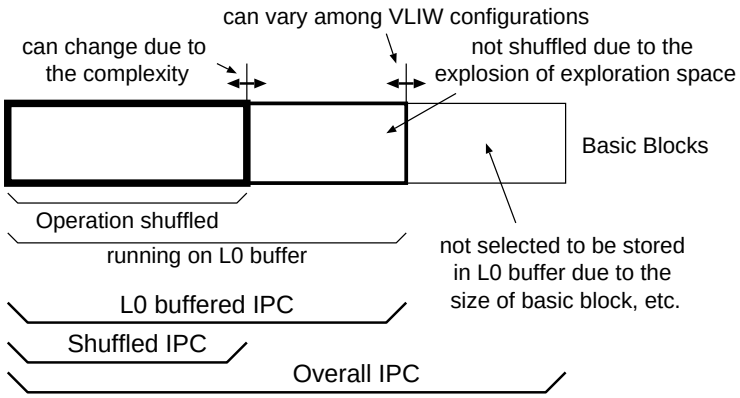


Figure 4.23: Various versions of IPC.

Table 4.9: Energy reduction for sha on 8-slot Hetero VLIW.

Initial Energy (mJ)	0.177
Maximum Energy (mJ)	0.195
Minimum Energy (mJ)	0.171
Reduction (Initial-Minimum)	3.4%
Reduction (Maximum-Minimum)	12.3%

(9 operations out of 10 slots), and consequently it leads to not so much energy reduction.

4.8 Conclusion

This chapter presented an optimization flow of L0 cluster generation on heterogeneous VLIW architectures and also proposes heuristics to decrease the size of the exploration space. Though L0 cluster generation has a potential gain of energy reduction of up to 63% as shown in [13], the experimental results of this section reveal that this gain can be improved furthermore up to 27.6% in 8 slots heterogeneous VLIW processor by using the proposed operation shuffling algorithm. On the assumption that an L0 buffer consumes 20 to 40% of total processor energy [8] [12], the reduction achieved by the proposed algorithm might look not so big, however, other processor components (ex. register files, data caches) can also be optimized for energy, as shown in [12]. Therefore, the author believes the approach has significant impact on energy efficiency. The results also show that a homogeneous architecture has more potential gain than a heterogeneous architecture. The proposed algorithm supports wide range of heterogeinity while the previous method [14] supports only limited range of heterogeinity. The experimental results also indicate that the proposed heuristics drastically reduce the exploration search space by about 90%, with comparable results, with differences of less than 1% on average, to full search.

Note that the proposed operation shuffling approach would be also applicable to other multi-dimensional architectures. In principle, the operation shuffling would be applied to array style of architectures like ADRES [61], a coarse-grained reconfigurable array architecture coupled with VLIW processor developed in IMEC, while the proposed method currently targets on a

linear (one-dimensional) style of VLIW processor. In such a multi-dimensional array processor, a similar approach to L0 cluster can be applied in order to reduce buffer access and activation of processing elements. Since the number of processing elements is, in general, not smaller than typical VLIW processors, clustering for coarse-grained control would also be important. Then a change of scheduling like the operation shuffling would be effective on energy reduction. To apply the proposed operation shuffling to other architecture, it would be needed to change a cost function which estimates energy for a given schedule and architecture. Then by a similar approach to the proposed operation shuffling can be utilized and a similar heuristic to the proposed heuristics would also be feasible.

Chapter 5

Efficient energy reduction method

The previous chapter describes an operation shuffling algorithm, which explores assignment of operations for each cycle, generates various schedules, and evaluates them to find an energy efficient schedule. This approach can find energy efficient schedules, however, it takes a long time to obtain the final result.

In this chapter, an efficient method to directly generate an energy efficient schedule without iterations of operation shuffling is described. In the proposed method, a compiler schedules operations using the result of the single operation shuffling as a constraint. This chapter also proposes some optimization algorithms to generate an energy efficient schedule for a given L0 cluster configuration. The proposed method can drastically reduce the computational effort since it performs the operation shuffling only once.

This chapter first analyzes the results of operation shuffling and then proposes an efficient method to directly generate an energy efficient schedule which can reduce the exploration space furthermore.

The experimental results show that comparable energy reduction can be achieved by using the proposed method while the computational effort can be reduced significantly over the conventional operation shuffling described in Chapter 4.

5.1 Problem and motivation

Figure 5.1 (a) shows an overview of an operation shuffling approach proposed in Chapter 4. An *initial schedule* is first obtained by compilation of the *application* on the specified *architecture* of target processor by using *retargetable VLIW C compiler*. The *initial schedule* is then further analyzed by a *schedule analyzer* which generates *activation information* of slot. Here the *schedule analyzer* generates all possible operation-shuffled schedules (i.e. *slot activation info*) for a basic block, according to the *architecture information* (target processor). Finally, an *L0 cluster optimizer* finds the most energy efficient L0 cluster configuration for each activation information, and reports an *optimized L0 cluster configuration* and *estimated energy* corresponding it. In this approach, the basic blocks are first ordered based on their weight (significance). Operation shuffling is first performed on the most significant basic block. The cost of previous basic blocks' shuffled schedule is kept into account while performing the shuffling

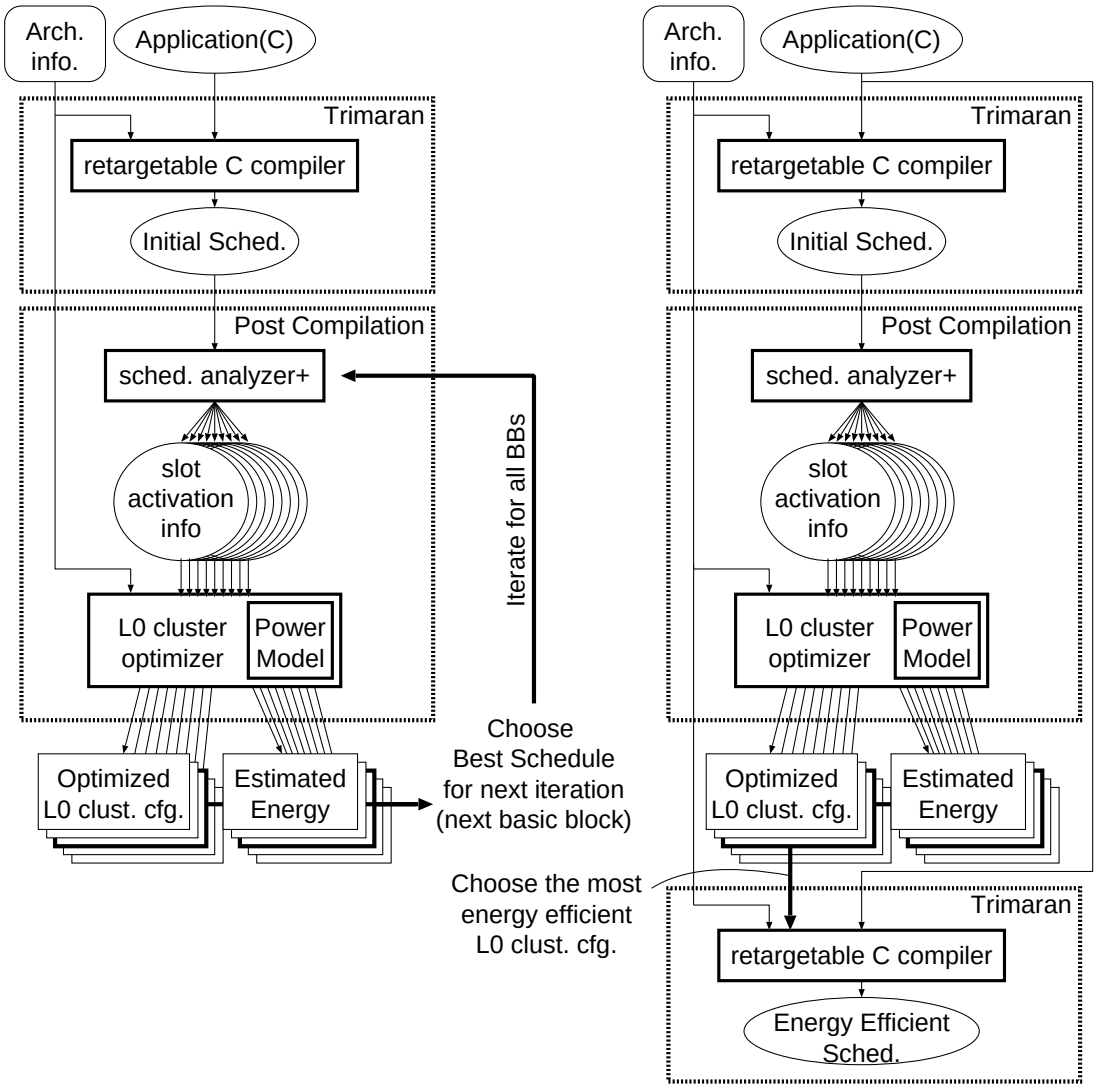


Figure 5.1: Overview of an L0 cluster configuration improvement phase (a) in the conventional way, (b) in the proposed method.

on the following basic block. Therefore, we can obtain the best schedule and optimized L0 cluster configuration after shuffling all basic blocks. Some heuristics are also proposed, which reduce the complexity of the search space by about 90%.

The approach can find energy efficient schedules; up to 30% of L0 buffer energy can be reduced as shown in Chapter 4. The approach, however, still takes a long time to obtain the final result, even if the heuristics are applied. in experiments, it usually takes thirty minutes but some large applications take a few days.

5.1.1 Analysis of existing operation shuffling approach

First the L0 cluster configuration obtained using the operation shuffling is evaluated. Here basic blocks are shuffled one by one for all basic blocks. We call this approach *shuffling all BBs*. In this experiments, three realistic kinds of VLIW processor targets are used:

1. 8 slot heterogeneous
2. 10 slot heterogeneous
3. 8 slot homogeneous

Table 4.1 and Table 4.2 show slot capability of 8 and 10 slot heterogeneous VLIW processors, respectively.

Table 5.1 shows the optimal L0 cluster configuration for some architectures and benchmarks, and the number of shuffled basic blocks needed to find the configuration. In Table 5.1, the third column shows the optimal cluster configuration obtained by the operation shuffling. Each number corresponds to a slot and represents the identification of L0 cluster. For example, ”01222222” represents the first and second slots form *L0 cluster 0* and *L0 cluster 1* respectively, and the rest of slots (from slot 3 to 8) form *L0 cluster 2*. The fourth column shows the number of shuffled basic blocks to find the cluster configuration. For example of adpcm decoder on 8 slot heterogeneous VLIW, the optimal cluster configuration was obtained after shuffling the second basic block. From this table, we can find that optimal cluster configurations are different for applications and architectures. This results also support the motivation to introduce the operation shuffling.

In Table 5.1, we see that after one basic block is shuffled, the optimal L0 cluster configuration is found in almost all cases. Even if more than one basic block is shuffled, the best cluster configuration is not changed.

5.2 Overview of the proposed method

This section describes a new more efficient method to generate an energy efficient schedule in a short time. The proposed method performs the operation shuffling for the most important basic block and considers the result of the shuffling as a constraint for other parts of the code. This makes the computational effort much smaller than the conventional approach which performs operation shuffling for all basic blocks.

Table 5.1: Optimal cluster configuration and the number of required basic blocks to find the configuration.

architecture	benchmark	cluster config.	#required BBs
8 hetero	adpcm decode	01222222	2
8 hetero	blowfish encode	01223323	1
8 hetero	cjpeg	01122222	1
8 hetero	epic	01122222	1
8 hetero	g721 decode	01122222	1
8 hetero	g721 encode	01222222	1
8 hetero	gsmencode	01122222	1
8 hetero	mpeg2 encode	00111111	1
8 hetero	sha	01223323	1
10 hetero	adpcm decode	0123100100	1
10 hetero	blowfish encode	0123401444	1
10 hetero	cjpeg	0123301000	1
10 hetero	epic	0123021000	50
10 hetero	g721 decode	0123300000	1
10 hetero	g721 encode	0123000010	4
10 hetero	gsmencode	0120311333	1
10 hetero	mpeg2 encode	0012200222	1
10 hetero	sha	0123401455	1
8 homo	adpcm decode	01222222	1
8 homo	blowfish encode	01112222	1
8 homo	cjpeg	01122222	1
8 homo	epic	00111122	1
8 homo	g721 decode	01122222	1
8 homo	g721 encode	01222222	1
8 homo	gsmencode	01122222	1
8 homo	mpeg2 encode	00112222	1
8 homo	sha	01223333	1

Figure 5.1 (b) shows an overview of the proposed method. In this method, it first applies the operation shuffling to the most significant basic block. Then the VLIW compiler runs again with the obtained L0 cluster configuration from the first operation shuffling. Since the optimal L0 cluster configuration is supposed to be found in the first operation shuffling, an energy efficient schedule can be obtained without further operation shuffling if the compiler can efficiently schedule for the given L0 cluster configuration as a constraint. So we now need an adapted version of the scheduling technique that can incorporate constraints.

5.3 Scheduling for a given L0 cluster configuration

This section proposes algorithms to change slot assignment of operations in the end of scheduling phase. A top-level description of the scheduling phases is outlined in Algorithm 2. An

Algorithm 2 Relevant phases in compiler back-end.

```
Schedule and allocate (block){
    Compute_Analysis_Info();
    Schedule_Ops(block);
    Register_Allocation();
    Reschedule_Operations();
}
```

application is translated into a control flow graph composed of *blocks*. Each block can be a basic block, hyper block or a super block, and scheduling is done one block at a time. Each block is annotated with analysis information like liveness and operation priorities in *Compute_Analysis_Info()*. In *Schedule_Ops()*, each operation is assigned to a certain cycle and a certain slot. Once the operations are scheduled, the data (variables, constants and other data structures) are allocated to registers in *Register_Allocation()*.

Reschedule_Operations() changes the assignment of operations after the register allocation phase. Here all optimization techniques have been applied and there is still enough freedom to change the operation assignment for improving the energy efficiency.

An outline of the rescheduling phase is shown in Algorithm 3.

5.3.1 Algorithm to try to fill an inefficient cluster

The first algorithm tries to move operations to a cluster that is used inefficiently. The inefficient cluster means a cluster that is not full but not empty; i.e. the cluster has to be activated in the cycle, however, there is a free slot in the cluster. Therefore, to fill the free slot with an operation is more power efficient if a cluster where the operation is originally assigned can be inactivated by this move.

Figure 5.2 (a) shows an example of this move. In this example, there are two L0 clusters, LC0 and LC1, and assume LC0 is lower power than LC1. Though both clusters have to be activated in the original assignment, LC0 can be inactivated since all operations assigned to LC0 can be moved to LC1. This move is energy efficient even if operations move to larger power cluster, since only one cluster needs to be activated after the move. Figure 5.2 (b), (c), and (d) are also the same kind of example. Algorithm 4 shows an algorithm to fill an inefficient cluster. In this algorithm, we search all clusters for an inefficient cluster. If an inefficient cluster is found, i.e. a cluster that is not empty but not fully filled, then we search for a cluster which can provide operations to fill the inefficient cluster and can be empty if it provides the operations. All conditions are fulfilled, then this algorithm tries to move operations to fill the inefficient cluster in *Move_Ops_bw_Clusters()*.

Algorithm 3 Reschedule operations between L0 clusters.

```
Reschedule_Operations(block){
  Build_Cluster_List(Clust);
  Calc_Cluster_Size(Clust, Size, NumSlot, Depth);
  if (block runs on L0 Buffer){
    Calc_ActiveCycle_in_Cluster(Clust, block, ActiveCycle);
    foreach (cycle in block){
      Sort_Cluster_by_Size(Size, Clust, ActiveCycle);
      Calc_FreeSlot_of_Cluster(Clust, cycle, block, FreeSlot);
      Try_To_Fill_Inefficient_Cluster(Clust, cycle, block, NumSlot,
                                      FreeSlot, ActiveCycle);
      Try_To_Move_To_Shallower_Cluster(Clust, cycle, block, NumSlot,
                                       FreeSlot, ActiveCycle);
      Try_To_Move_To_Wider_Cluster(Clust, cycle, block, NumSlot,
                                   FreeSlot, ActiveCycle);
    }}}
```

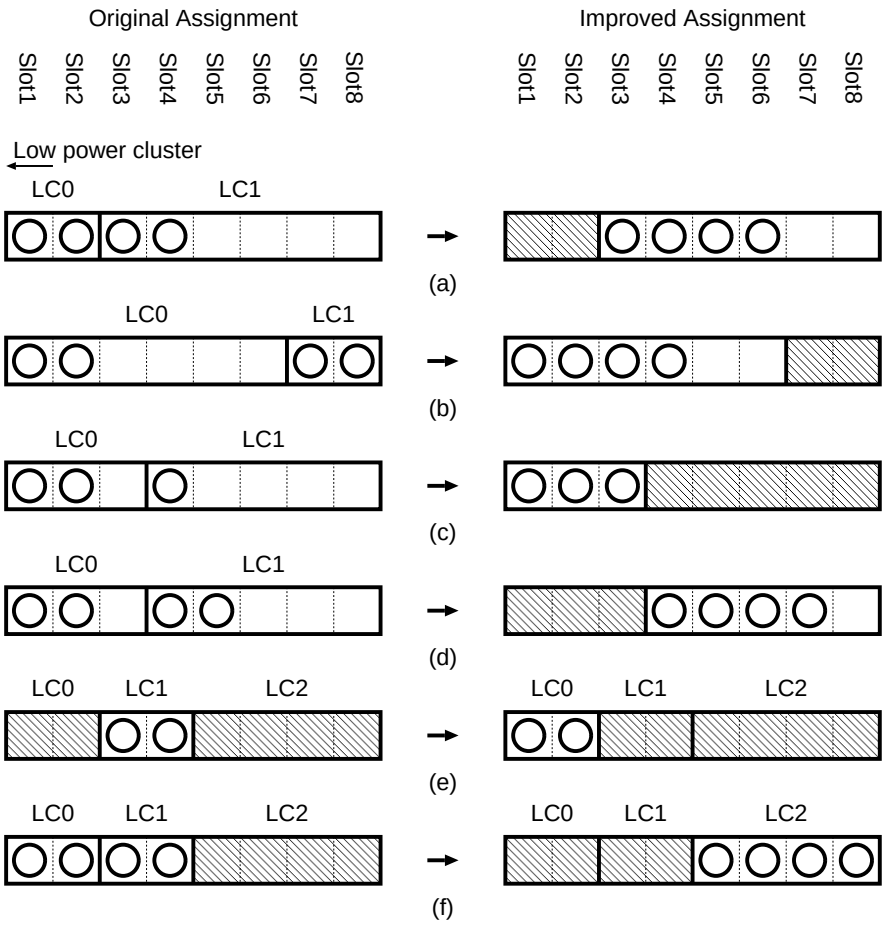


Figure 5.2: Examples of rescheduling algorithm.

Algorithm 4 Try to fill an inefficient cluster.

```
Try_To_Fill_Inefficient_Cluster(Clust, cycle, block, NumSlot, FreeSlot, ActiveCycle){
  for (h = 0 .. Num_Clust-1) {
    if (FreeSlot[h] > 0 and FreeSlot[h] ≠ NumSlot[h]){
      for (j = Num_Clust-1 .. 0) {
        if (j = h) continue;
        if (FreeSlot[h] ≥ (NumSlot[j] - FreeSlot[j])){
          if (ActiveCycle[h] ≥ Depth[h]){
            // do nothing
          } else {
            moved = Move_Ops_bw_Clusters(cycle, Clust, j, h, block);
            if (moved > 0) {
              if (FreeSlot[h] = NumSlot[h]) ActiveCycle[h]++;
              FreeSlot[h] -= moved;
              FreeSlot[j] += moved;
              if (FreeSlot[j] = NumSlot[j]) ActiveCycle[j]--;
              if (FreeSlot[h] = 0) break;
            }}}}}}}
```

5.3.2 Algorithm to try to move operations to a shallower cluster

The second optimization algorithm tries to move operations to a shallower cluster. Here a shallower cluster means a cluster with the same width but with shallower depth (i.e. low power cluster). Even if there is no more inefficient cluster, by moving operations from a larger cluster to a smaller cluster, the energy efficiency can be improved furthermore. Figure 5.2 (e) shows an example of this case. In this example, there is no inefficient cluster any more; i.e. all clusters are empty or completely filled with operations. However, LC1 has to be activated while a smaller cluster LC0 is inactivated. By moving all operations assigned in LC1 to LC0, the larger power cluster LC1 can be inactivated instead of LC0. Algorithm 5 shows an algorithm to move operations to such a shallower cluster. In this algorithm, it first searches for an empty cluster. Then for the cluster the algorithm searches for a larger cluster which contains operations. If a cluster which satisfies all conditions is found, the algorithm tries to move operations using *Move_Ops_bw_Clusters()*.

Note that the algorithm uses *ActiveCycle* to keep track of the number of activated cycles for each cluster in order to avoid exceeding the depth of given cluster configuration. It sometimes restricts the freedom of rescheduling, however, extreme concentration on a single cluster makes the depth of L0 cluster deeper and consequently leads increase of energy.

5.3.3 Algorithm to try to move operations to a wider cluster

As the third optimization, this section proposes an algorithm that tries to move operations to a wider cluster. It could happen that the total power of clusters which are full with operations is larger than the power of an empty cluster. Figure 5.2 (f) shows this case. In this example,

Algorithm 5 Try to move operations to a shallower cluster.

```
Try_To_Move_To_Shallower_Cluster(Clust, cycle, block, NumSlot, FreeSlot, ActiveCycle){
  for (h = 0 .. Num_Clust-1) {
    if (FreeSlot[h] = NumSlot[h]) { /* empty */
      for (j = Num_Clust-1 .. h+1) {
        if (NumSlot[j] = NumSlot[h] and Size[j] > Size[h]){
          if (FreeSlot[j] = 0) {
            if (ActiveCycle[h] ≥ Depth[h]){
              // do nothing
            } else {
              moved = Move_Ops_bw_Clusters(cycle, Clust, j, h, block);
              if (moved > 0) {
                if (FreeSlot[h] = NumSlot[h]) ActiveCycle[h]++;
                FreeSlot[h] -= moved;
                FreeSlot[j] += moved;
                if (FreeSlot[j] = NumSlot[j]) ActiveCycle[j]--;
                break;
              }}}}}}}}
```

LC0 and LC1 are full with operations and LC2 is completely empty. Then there is no further improvement for the last two optimization algorithms. We assume the power of LC0 and LC1 (P_0 and P_1, respectively) is less than the power of LC2 (P_2). However, if the total power of LC0 and LC1 is larger than LC2 (i.e. $P_0 + P_1 > P_2$), moving all operations to LC2 is beneficial. Algorithm 6 shows an algorithm to selectively move operations to a wider cluster. This algorithm first searches for an empty cluster. Then for that cluster it makes a combination of full clusters whose number of operations is less than the width of the empty cluster. If the total power of clusters is larger than the empty cluster, it tries to move operations to the empty cluster. Apparently this problem to make a combination of clusters which fits the limit of width and power is known as a knapsack problem, and consequently it is an NP hard problem. The proposed non greedy technique does not care about a *slot capability* when making a combination of clusters; some operations might not move to the new cluster in case that no slot in the cluster can issue the operations. Hence, the proposed algorithm does not try to pursue an optimum combination. However, it yields enough quality of solution with much reduced computational effort.

5.4 Experimental Results

This section describes experimental results, which show the applicability of the proposed method.

Table 5.2 shows the comparison of the proposed method and conventional methods for one benchmark (g721 decoder on 8 slot homogeneous VLIW). *Monolithic L0* uses a monolithic L0 cluster with initial schedule. In *Initial*, L0 clusters are generated for an initial schedule. *Shuffling all BBs* shuffles all basic blocks, and *Shuffling 1 BB* is a result after shuffling only one

Algorithm 6 Try to move operations to a wider cluster.

```
Try_To_Move_To_Wider_Cluster(Clust, cycle, block, NumSlot, FreeSlot, ActiveCycle){
  for (h = 0 .. Num_Clust-1) {
    if (FreeSlot[h] = NumSlot[h]) { /* h is empty */
      power_h = size[h];
      power_mov = 0;
      MOV = ∅;
      for (j = Num_Clust-1 .. 0) {
        if (j = h) continue;
        if (NumSlot[j] < NumSlot[h]){
          power_j = size[j];
          ops_j = width[j] - free[j];
          if (FreeSlot[j] = 0) {
            if (Free[h] ≥ |MOV| + ops_j){
              power_mov += power_j;
              MOV += j;
            }}}}
      if (power_mov > power_h){
        if (ActiveCycle[h] ≥ Depth[h]){
          // do nothing
        } else {
          for (each k in MOV) {
            moved = Move_Ops_bw_Clusters(cycle, Clust, k, h, block);
            if (moved > 0) {
              if (FreeSlot[h] = NumSlot[h]) ActiveCycle[h]++;
              FreeSlot[h] -= moved;
              FreeSlot[k] += moved;
              if (FreeSlot[k] = NumSlot[k]) ActiveCycle[k]--;
            }}}}}}}
```

Table 5.2: Comparison of estimated energy (g721 decoder@8 slot Homo).

Method	Energy (mJ)	(%)	Exploration Effort	Approx. Time (sec)
Monolithic L0	2.8912	312.0%	c	20
Initial	0.9268	100.0%	c	20
Shuffling 1 BB	0.9021	97.3%	224s+c	30
Shuffling all BBs	0.7815	84.3%	60696s+c	2600
Proposed	0.8003	86.3%	224s+2c	50

basic block. In *Proposed*, operation shuffling is performed for only the first basic block and operations are re-scheduled for the obtained L0 cluster configuration. The second column shows the estimated energy for a optimal solution obtained by each method and the third column provides the relative energy over *Initial*. In the fourth column, exploration effort is represented; c is computational effort of single compilation and s is effort of evaluating a candidate in operation shuffling. In this experiment, compilation for this example takes 20 seconds while shuffling of 60696 patterns takes 43 minutes; i.e. c is 20 second and s is 0.043 second. The fifth column shows approximate exploration time calculated using the c and s. Note that the experimental environment is Fedora Core 3 on Pentium 4 3.2GHz with 4096MB memory.

Though the energy of *Proposed* is little worse than *Shuffling all Bbs* (only 2%), the proposed method significantly reduces the exploration space; it is about 50 times faster than *Shuffling all BBs*. The proposed method takes almost the same exploration time as the mere greedy *Shuffling 1 BB*. However, it clearly achieves less energy (factor of 11%).

Compared with *Monolithic L0*, *Initial* achieves less energy consumption, however, the result shows that *Initial* is not so energy efficient without the proposed method nor operation shuffling.

Figure 5.3 shows a comparison of energy for various combinations of benchmark application and processor architecture. In most of cases, the proposed method achieves less energy than *Shuffling 1 BB*, and smaller than even *Shuffling all BBs* in some cases. The reason why the proposed method surpasses *Shuffling all BBs* would be because of the heuristics used in *Shuffling all BBs*; an optimal schedule might be omitted in *Shuffling all BBs*, while the proposed scheduling algorithm has a chance to generate an optimal schedule that is missed by the heuristics.

On the contrary, in some combinations the proposed method yields not so good result. This result can be seen in Fig. 5.3 where the results of most of benchmarks on 10 slot heterogeneous VLIW show the proposed method does not handle energy gain sufficiently. This is because moving operations to another cluster is difficult due to the *slot capability* in heterogeneous VLIW processors even if there is a free slot in the destination cluster, as discussed in Sec. 5.3. The architecture of 10 slot heterogeneous VLIW that is used in this experiment has limited capability for a slot, as shown in Table 4.2.

Furthermore, energy reduction is not obtained in a case where the first operation shuffling does not yield the optimal cluster configuration (see Table 5.1). This happens in epic on the 10 slot heterogeneous VLIW. Since the rescheduling algorithm is performed on a not optimal cluster configuration, the generated schedule is not so energy efficient. These might be a limitation

of the proposed method. However, in most of the cases the results support the feasibility of the proposed method; the energy reduction of the proposed method is notable compared with other scalable methods. The average energy reduction of the proposed method over *Shuffling 1 BB* is 11% in 8 slot heterogeneous and 9% in 8 slot homogeneous VLIW processors.

5.5 Conclusion

This chapter presented an efficient method to generate an energy efficient schedule. Based on the analysis of characteristics of energy efficient L0 cluster configuration obtained from the operation shuffling, it is found that the optimal L0 cluster configuration is fixed after the first iteration of operation shuffling. Therefore, in the proposed method, the operation shuffling is performed only once for the most significant basic block and a compiler schedules again for the obtained cluster configuration. The proposed method combines a compiler technique of scheduling for a given L0 cluster configuration with an operation shuffling framework. Some algorithms to schedule for a given cluster configuration are proposed. The proposed method can drastically reduce the computational effort by a factor of 50, hence it improves the design productivity of low energy embedded systems.

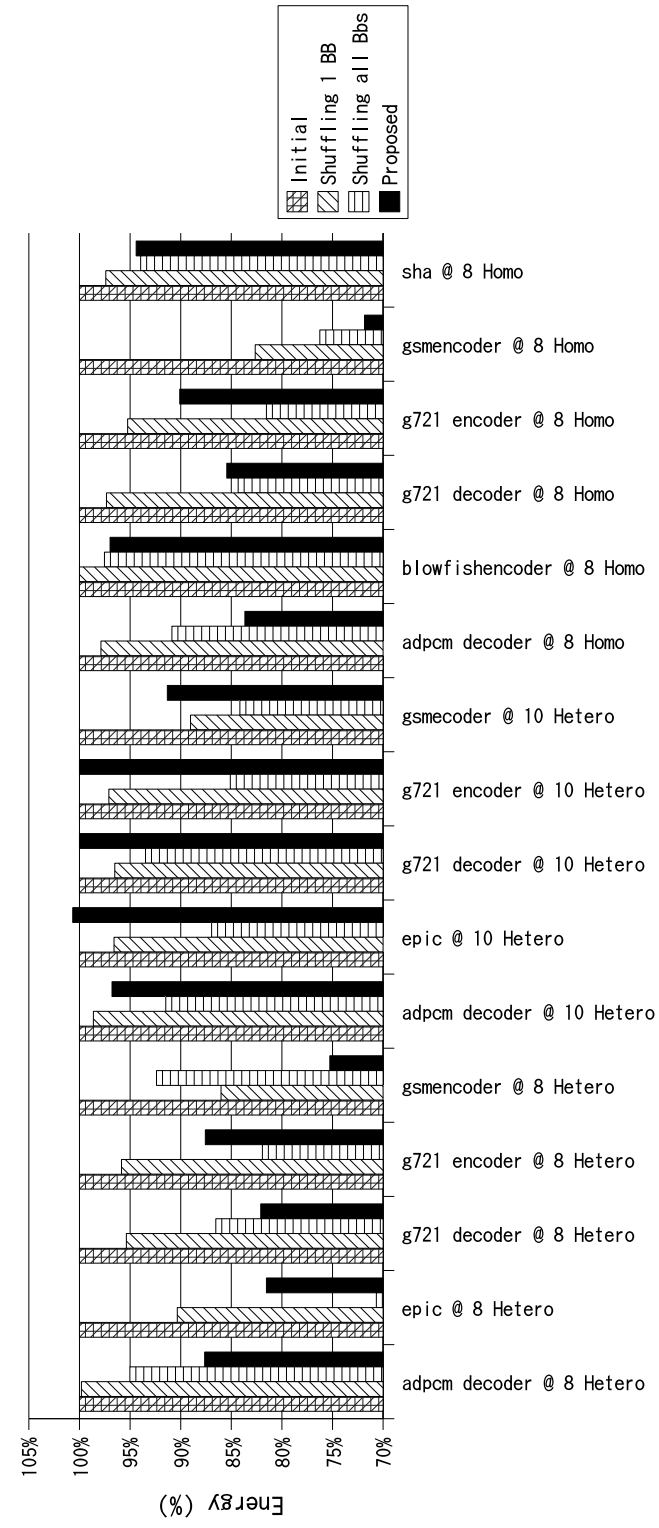


Figure 5.3: Comparison of energy reduction.

Chapter 6

Conclusion and future work

In this chapter, the conclusion of this thesis and the future work are described.

6.1 Conclusion

This thesis describes a low power design method for embedded systems using VLIW processor.

VLIW processors are known as an effective solution for embedded systems which require both of high performance and low energy, however, there are a lot of architectural parameters to be decided by designers. Since these parameters significantly affect the performance and area, it is required to perform the design space exploration where designers evaluate many architectures to determine the optimal parameter set. However, designing a VLIW processor was very complex, and consequently time consuming and error-prone. Hence, design space exploration on VLIW processors could not have been performed sufficiently so far.

The first part of this thesis describes a VLIW processor generation method. Chapter 3 proposes a synthesizable HDL generation method for configurable VLIW processors, which supports a flexible architecture model, especially in dispatching rules. Experimental results shows that the proposed method can generate a VLIW processor from a high level description, which is 80% to 90% smaller than HDL description. And also, the generation time of HDL description is sufficiently short, that is from 2 to 15 seconds. Since the specification description supports a wide range of dispatching rules and the amount of description is sufficiently small, it is possible to generate a wide range of fine-quality VLIW processors in a short time. Note that a simple *copy and paste* strategy can be employed during preparation of the processor specification description. Hence, the actual effort that designers have to describe is much smaller than the manual design of HDL. Though a generated VLIW processor has not been compared with a manually designed VLIW processor yet, it is assumed that the quality of generated HDL description is almost the same as that of manually designed HDL description as discussed in Section 3.7.1.2. Therefore, the proposed method can significantly improve the design productivity of VLIW processors. This work is reported in [1] and [2].

The second part of this thesis discusses a low power method for VLIW processors. The energy breakdown of VLIW processors indicates that the power bottleneck of VLIW processors is in the instruction memory hierarchy (e.g. instruction fetch). An L0 buffer and an L0 cluster architecture have been proposed to reduce the energy in the instruction memory hierarchy.

However, the result of L0 cluster generation is sensitive to the schedule of the target application.

Chapter 4 describes an operation shuffling algorithm for improvement of energy efficiency of L0 cluster. Since an L0 cluster configuration is very sensitive to operation scheduling, various schedules are generated and evaluated in order to obtain an optimal schedule. In the proposed algorithm, by shuffling all basic blocks iteratively, energy consumption can be reduced significantly. To reduce the size of the exploration space, some heuristics are also described in Chapter 4. The experimental results show that the proposed operation shuffling algorithm successfully reduces energy consumption in various VLIW processors including heterogeneous VLIW processors as well as homogeneous VLIW processors. This work is reported in [3] and [4].

Since the simple operation shuffling takes huge amount of time even if the above heuristics are applied, a more efficient method to find a low energy operation schedule is described in Chapter 5. Based on the analysis of characteristics of energy efficient L0 cluster configuration obtained from the operation shuffling, it is found that the optimal L0 cluster configuration is fixed after the first iteration of operation shuffling. Therefore, in the proposed method, the operation shuffling is performed only once for the most significant basic block and a compiler schedules again for the obtained cluster configuration. Some algorithms to schedule for a given cluster configuration are described in Chapter 5. By exploiting the scheduling algorithms, a compiler can generate a low energy schedule in a straightforward way. The experimental result shows that the proposed method can generate energy efficient schedules with 50 times shorter exploration time.

Figure 6.1 shows a contribution of the proposed operation shuffling method. As discussed in Section 4.1 using Fig. 4.2, the instruction memory hierarchy was still a power bottleneck after applying some conventional optimization algorithms. The L0 cluster reduces the energy by up to 67% and the proposed operation shuffling improves it furthermore by about 30%. Then the total energy is 62% smaller than the energy before optimizations. In Fig. 6.1, the contribution of operation shuffling might look not so notable, however, other processor components such as the data memory and the data path will also be optimized furthermore as indicated in [62]. Then the instruction memory hierarchy needs to be optimized again and the proposed method acts the significant role in the energy efficiency.

6.2 Future work

The future work includes the following items.

6.2.1 Future work on VLIW synthesis

The input of the VLIW processor generation method is in higher level than RTL, however, it still requires complicated description. Especially, determining and describing resource groups that are required for a target architecture is tedious and troublesome work. A simpler input description helps a designer and improves the design productivity furthermore.

And also, the quality of generated VLIW processor would be improved furthermore. The decoding logic now is large and complicated. It is because the proposed method supports

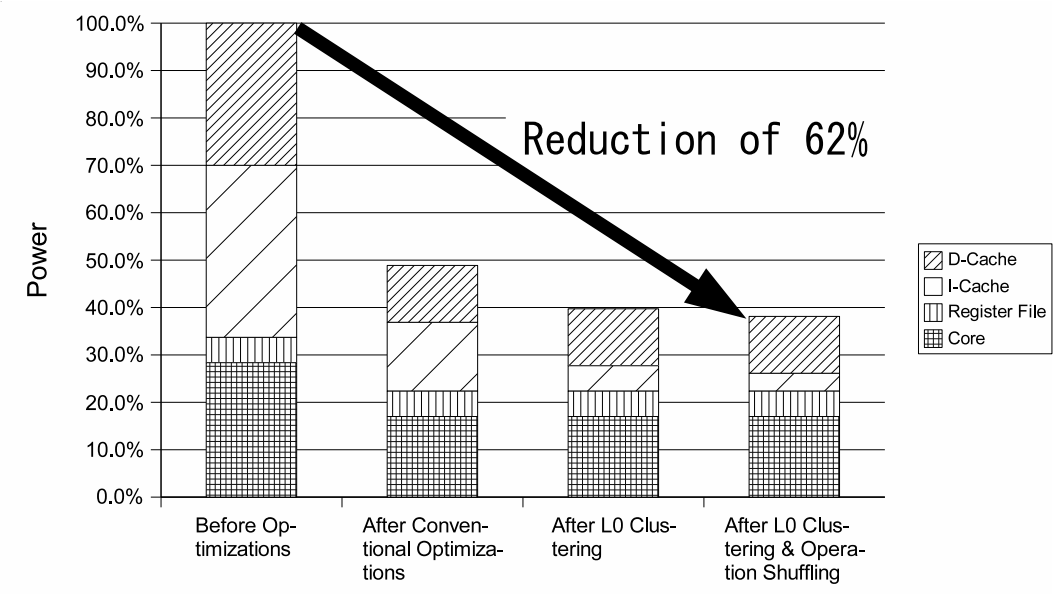


Figure 6.1: Power reduction by the proposed method.

a wide variety of dispatching rules. However, when a dispatching rule is not complex, for instance, no FU sharing, the decoding and dispatching logic can be simplified. Simpler logic is very beneficial in terms of area, delay time, and power consumption.

6.2.2 Future work on operation shuffling

The future work on operation shuffling includes operation shuffling across cycle boundaries. An algorithm proposed in this thesis only moves an operation within cycle boundaries. The algorithm already yields a good result as shown in this thesis, however, it can be improved furthermore by allowing move across the cycle boundaries. Some operations can move to another cycle unless the data dependency between related operations is violated. Hence, there would be more opportunities to optimize operation scheduling for energy efficiency.

The shuffling over the cycle boundaries should achieve more energy gain. The complexity of the shuffling over the cycle boundaries will, however, become too huge since an operation has more freedom to move not only to different slots but also to different cycles. To manage this problem, another new and different algorithm (heuristic) is needed to efficiently prune the exploration space.

In principle, the proposed heuristics, which limit the exploration space during operation shuffling within a cycle, can be utilized for exploration space reduction of shuffling over the cycle boundaries. Intuitively, it is a realistic assumption that an operation can also move to a different issue-slot when it moves to another cycle. Then, after moving to different cycles, the same technique for shuffling within a cycle can be used. Hence, this thesis started with the topic of shuffling within a cycle.

A preliminary experiment shows a result supporting a prospect that the shuffling over cycle boundaries yields more energy gain; for example in a case, the energy is reduced by 10.3%

by the shuffling over cycle boundaries, while the shuffling within a cycle only yields 4.9% of energy reduction in the same case.

Further improvement in the effectiveness of shuffling over the cycle boundaries would be achieved with a software multi-threading approach would be beneficial, which combines multiple independent loops into one loop to improve performance [60]. Since independent loops are integrated into one loop, each operation has less dependency for each other in the loop. This gives an operation more freedom to move to different cycles and consequently more opportunity for energy reduction.

Another category of future work is operation shuffling on other architectures. In principle, the proposed operation shuffling would be applied to array style of architectures like ADRES [61], a coarse-grained reconfigurable array architecture coupled with VLIW processor developed in IMEC, while the current target of the proposed method is a linear (one-dimensional) style of VLIW processor. In such a multi-dimensional array processor, a similar approach to L0 cluster can be applied in order to reduce buffer access and activation of processing elements. Since the number of processing elements is, in general, not smaller than typical VLIW processors, clustering for coarse-grained control would also be important. Then a change of scheduling like the operation shuffling would be effective on energy reduction. To apply the proposed operation shuffling to other architecture, it would be needed to change a cost function which estimates energy for a given schedule and architecture. Then by a similar approach to the proposed operation shuffling can be utilized and a similar heuristic to the proposed heuristics would also be feasible.

Bibliography

[1] Y. Kobayashi, S. Kobayashi, K. Okuda, K. Sakanushi, Y. Takeuchi, and M. Imai, "Synthesizable HDL generation method for configurable VLIW processors," in Proc. Asia and South Pacific Design Automation Conference (ASP-DAC), pp.843–846, Jan. 2004.

[2] Y. Kobayashi, S. Kobayashi, K. Sakanushi, Y. Takeuchi, and M. Imai, "HDL generation method for configurable VLIW processor," IPSJ Journal, vol.45, no.5, pp.1311–1321, May 2004. (in Japanese).

[3] Y. Kobayashi, M. Jayapala, P. Raghavan, F. Catthoor, and M. Imai, "Operation shuffling for low energy l0 cluster generation on heterogeneous VLIW processors," in Proc. IEEE 3rd Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia 2005), pp.81–86, Sept. 2005.

[4] Y. Kobayashi, M. Jayapala, P. Raghavan, F. Catthoor, and M. Imai, "Methodology for operation shuffling and l0 cluster generation for low energy heterogeneous VLIW processors," ACM Trans. on Design Automation of Electronic Systems. (to appear).

[5] J. Ganssle and M. Barr, Embedded Systems Dictionary, CMP Books, 600 Harrison Street, San Francisco, CA 94107 USA, 2003.

[6] M. Johnson, Superscalar Microprocessor Design, Prentice-Hall, Inc., 1991.

[7] J.A. Fisher, "Very Long Instruction Word Architectures and the ELI-512," in Proc. the 10th Annual Symposium on Computer Architectures, pp.140–150, 1983.

[8] L. Benini, D. Bruni, M. Chinosi, C. Silvano, V. Zaccaria, and R. Zafalon, "A power modeling and estimation framework for VLIW-based embedded systems," in Proc. IEEE International Workshop on Power And Timing Modeling, Optimization and Simulation (PATMOS), IEEE, Sept. 2001.

[9] L.H. Lee, B. Moyer, and J. Arends, "Instruction fetch energy reduction using loop caches for embedded applications with small tight loops," in Proc. Int'l Symp. on Low Power Electronic Design (ISLPED), pp.267–269, Aug. 1999.

[10] R.S. Bajwa, M. Hiraki, H. Kojima, D.J. Gorny, K. Nitta, A. Shridhar, K. Seki, and K. Sasaki, "Instruction buffering to reduce power in processors for signal processing," IEEE Trans. VLSI Syst., vol.5, no.4, pp.417–424, Dec. 1997.

[11] N. Bellas, I. Hajj, C. Polychronopoulos, and G. Stamoulis, “Architectural and compiler support for energy reduction in the memory hierarchy of high performance microprocessors,” in Proc. Int’l Symp. on Low Power Electronic Design (ISLPED), Aug. 1998.

[12] A. Lambrechts, P. Raghavan, A. Leroy, G. Talavera, T. Vander Aa, M. Jayapala, F. Catthoor, D. Verkest, G. Deconinck, H. Coporaal, F. Robert, and J. Carrabina, “Power breakdown analysis for a heterogeneous NoC platform running a video application,” in Proc. IEEE 16th International Conference on Application-specific Systems, Architectures and Processors (ASAP), pp.179–184, July 2005.

[13] M. Jayapala, F. Barat, T. Vander Aa, F. Catthoor, H. Corporaal, and G. Deconinck, “Clustered loop buffer organization for low energy VLIW embedded processors,” IEEE Trans. Computers, vol.54, no.6, pp.672–683, June 2005.

[14] M. Jayapala, T. Vander Aa, F. Barat, F. Catthoor, H. Coporaal, and G. Deconinck, “L0 cluster synthesis and operation shuffling,” in Proc. IEEE International Workshop on Power And Timing Modeling, Optimization and Simulation (PATMOS), pp.311–321, IEEE, Sept. 2004.

[15] T. Pering, T. Burd, and R. Brodersen, “The simulation and evaluation of dynamic voltage scaling algorithms,” in Proc. Int’l Symp. Low Power Electronics and Design (ISLPED), pp.76–81, Aug. 1998.

[16] F. Catthoor, F. Balasa, E.D. Greef, and L. Nachtergaele, Custom Memory Management Methodology: Exploration of Memory Organization for Embedded Multimedia System Design, Kluwer Academic Publisher, 1998.

[17] W. Tang, R. Gupta, and A. Nicolau, “Power savings in embedded processors through decode filter cache,” in Proc. Design Automation and Test in Europe (DATE), March 2002.

[18] G.R. Uh, Y. Wang, D. Whalley, S. Jinturkar, C. Burns, and V. Cao, “Effective exploitation of a zero overhead loop buffer,” LCTES ’99: Proc. the ACM SIGPLAN 1999 workshop on Languages, compilers, and tools for embedded systems, pp.10–19, ACM Press, 1999.

[19] J.W. Sias, H.C. Hunter, and W. mei W. Hwu, “Enhancing loop buffering of media and telecommunications applications using low-overhead predication,” in Proc. 34th Annual Int’l Symp. on Microarchitecture (MICRO), Dec. 2001.

[20] G. Goossens, J. Van Praet, D. Lanneer, W. Geurts, A. Kifli, C. Liem, and P.G. Paulin, “Embedded software in real-time signal processing systems: Design technologies,” Proc. IEEE, vol.85, no.3, pp.436–454, March 1997.

[21] J. Sato, A.Y. Alomary, Y. Honma, T. Nakato, A. Shiomi, N. Hikichi, and M. Imai, “PEAS-I: A hardware/software codesign system for ASIP development,” IEICE Trans. Fundamentals, vol.E77-A, no.3, pp.483–491, Mar. 1994.

[22] J.H. Yang, B.W. Kim, S.J. Nam, J.H. Cho, S.W. Seo, C.H. Ryu, *et al.*, "MetaCore: An application specific DSP development system," in Proc. Design Automation Conference (DAC), pp.800–803, June 1998.

[23] J. Yang, B. Kim, S. Nam, Y. Kwon, D. Lee, J. Lee, C. Hwang, Y. Lee, S. Hwang, I. Park, and C. Kyung, "MetaCore: An Application-Specific Programmable DSP Development System," IEEE Trans. VLSI Syst., vol.8, no.2, pp.173–183, April 2000.

[24] G. Ezer, "Xtensa with user defined DSP coprocessor microarchitectures," in Proc. 2000 IEEE International Conference on Computer Design: VLSI in Computers & Processors, pp.335–342, Sept. 2000.

[25] G. Hadjiyiannis, S. Hanono, and S. Devadas, "ISDL: An instruction set description language for retargetability and architecture exploration," Design Automation for Embedded Systems, vol.6, no.1, pp.39–69, Sept. 2000.

[26] G. Hadjiyiannis, S. Hanono, and S. Devadas, "ISDL: A instruction set description language for retargetability," in Proc. Design Automation Conference (DAC), pp.299–302, June 1997.

[27] G. Hadjiyiannis, P. Russo, and S. Devadas, "A methodology for accurate performance evaluation in architecture exploration," in Proc. Design Automation Conference (DAC), pp.927–932, June 1999.

[28] A. Fauth, J. Van Praet, and M. Freericks, "Describing instruction set processors using nml," in Proc. European Design and Test Conference, pp.503–507, March 1995.

[29] J. Van Praet, G. Goossens, D. Lanneer, and H. De Man, "Instruction set definition and instruction selection for asips," in Proc. 7th IEEE Int. Symp. on High-Level Synthesis, May 1994.

[30] A. Hoffmann, T. Kogel, A. Nohl, G. Braun, O. Schliebusch, O. Wahlen, A. Wieferink, and H. Meyr, "A novel methodology for the design of application-specific instruction-set processors (ASIPs) using a machine description language," IEEE Trans. Computer-Aided Design, vol.20, no.11, pp.1338–1354, Nov. 2001.

[31] S. Pees, A. Hoffmann, V. Zivojnovic, and H. Meyr, "LISA - machine description language for cycle-accurate models of programmable DSP architecture," in Proc. Design Automation Conference (DAC), pp.933–938, June 1999.

[32] A. Hoffmann, H. Meyr, and R. Leupers, Architecture Exploration for Embedded Processors with LISA, Kluwer Academic Publishers, Boston, 2002.

[33] P. Mishra, A. Kejariwal, and N. Dutt, "Rapid exploration of pipelined processors through automatic generation of synthesizable RTL models," in Proc. 14th IEEE International Workshop on Rapid Systems Prototyping, pp.226–232, June 2003.

[34] P. Grun, A. Halambi, N. Dutt, and A. Nicolau, "RTGEN – an algorithm for automatic generation of reservation tables from architectural descriptions," IEEE Trans. VLSI Syst., vol.11, no.4, pp.731–737, Aug. 2003.

[35] P. Mishra, A. Kejariwal, and N. Dutt, "Synthesis-driven exploration of pipelined embedded processors," in Proc. International Conference of VLSI Design, pp.921–926, Jan. 2004.

[36] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau, "EXPRESSION: A language for architecture exploration through compiler/simulator retargetability," in Proc. Design, Automation and Test in Europe (DATE), pp.485–490, March 1999.

[37] M. Kandemir, M.J. Irwin, G. Chen, and I. Kolcu, "Compiler-guided leakage optimization for banked scratch-pad memories," IEEE Trans. VLSI Syst., vol.13, no.10, pp.1136–1146, Oct. 2005.

[38] Texas Instruments, "TMS320C6000 CPU and instruction set reference guide," Oct. 2000.

[39] Silicon Hive. http://www.silicon-hive.com/.

[40] Clear Speed. http://www.clearspeed.com/.

[41] P. Faraboschi, G. Brown, J.A. Fisher, G. Desoli, and F. Homewood, "Lx: A technology platform for customizable VLIW embedded processing," in Proc. Int'l Symp. on Computer Architecture (ISCA), pp.203–213, June 2000.

[42] A. Bona, M. Sami, D. Sciuto, C. Silvano, V. Zaccaria, and R. Zafalon, "Energy estimation and optimization of embedded VLIW processors based on instruction clustering," in Proc. Design Automation Conference (DAC), pp.886–891, June 2002.

[43] A. Bona, M. Sami, D. Sciuto, C. Silvano, V. Zaccaria, and R. Zafalon, "An instruction-level methodology for power estimation and optimization of embedded VLIW cores," in Proc. Design, Automation and Test in Europe (DATE), p.1128, March 2002.

[44] K. Okuda, S. Kobayashi, Y. Takeuchi, and M. Imai, "A simulator generator based on configurable VLIW model considering synthesizable HW description and SW tools generation," in Proc. the Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI), pp.152–159, April 2003.

[45] T. Maeda, J. Sato, Y. Takeuchi, and M. Imai, "A generation method for an interrupt controller in application specific instruction-set processor design," Technical Report of IEICE, VLD2001-118, vol.101, no.468, pp.39–44, Nov. 2001. (in Japanese).

[46] M. Itoh, Y. Takeuchi, M. Imai, and A. Shiomi, "Synthesizable HDL generation for pipelined processors from a micro-operation description," IEICE Trans. on Fundamentals of Electronics Communications and Computer Sciences, vol.E83-A, no.3, pp.394–400, March 2000.

[47] M. Itoh, A. Shiomi, J. Sato, Y. Takeuchi, and M. Imai, “Processor generation method for pipelined processors in consideration with pipeline hazards,” IPSJ Journal, vol.41, no.4, pp.851–862, Apr. 2000. (in Japanese).

[48] A. Kitajima, M. Itoh, J. Sato, A. Shiomi, Y. Takeuchi, and M. Imai, “Effectiveness of the ASIP design system PEAS–III in design of pipelined processors,” in Proc. Asia and South Pacific Design Automation Conference (ASP-DAC), pp.649–654, Feb. 2001.

[49] J.L. Hennessy and D.A. Patterson, Computer Architecture: A Quantitative Approach, Morgan Kaufmann Publishers, Inc., California, 1990.

[50] J. Monteiro, S. Devadas, P. Ashar, and A. Mauskar, “Scheduling techniques to enable power management,” in Proc. Design Automation Conference (DAC), pp.349–352, June 1996.

[51] N. Vijaykrishnan, M. Kandemir, M.J. Irwin, H.S. Kim, W. Ye, and D. Duarte, “Evaluating integrated hardware-software optimizations using a unified energy estimation framework,” IEEE Trans. Comput., vol.52, no.1, pp.59–76, Jan. 2003.

[52] T. Vander Aa, M. Jayapala, F. Barat, G. Deconinck, R. Lauwereins, F. Catthoor, and H. Coporaal, “Instruction buffering exploration for low energy VLIW with instruction clusters,” in Proc. IEEE Asia and South Pacific Design Automation Conference (ASP-DAC), pp.825–830, IEEE, Jan. 2004.

[53] D. Brooks, V. Tiwari, and M. Martonosi, “Wattch: A framework for architectural-level power analysis and optimizations,” in Proc. Int’l Symp. on Computer Architecture (ISCA), pp.83–94, June 2000.

[54] P.O. de Beeck, F. Barat, M. Jayapala, and R. Lauwereins, “CRISP: A template for reconfigurable instruction set processors,” in Proc. International Conference on Field Programmable Logic and Applications, pp.296–305, Aug. 2001.

[55] Trimaran, “Trimaran: An infrastructure for research in instruction-level parallelism.” http://www.trimaran.org/.

[56] A. Gordon-Ross and F. Vahid, “Frequent loop detection using efficient nonintrusive on-chip hardware,” IEEE Trans. Comput., vol.54, no.10, pp.1203–1215, Oct. 2005.

[57] D.C. Suresh, W.A. Najjar, F. Vahid, J.R. Villarreal, and G. Stitt, “Profiling tools for hardware/software partitioning of embedded applications,” in Proc. Language, Compiler and Tool Support for Embedded Systems (LCTES ’03), pp.189–198, June 2003.

[58] S. Rixner, W.J. Dally, B. Khailany, P. Mattson, U.J. Kapasi, and J.D. Owens, “Register organization for media processing,” in Proc. Int’l Symp. on High-Performance Computer Architecture (HPCA6), pp.375–386, Jan. 2000.

[59] MediaBench. http://cares.icsl.ucla.edu/MediaBench/.

[60] D.P. Scarpazza, P. Raghavan, D. Novo, F. Catthoor, and D. Verkest, “Software simultaneous multi-threading, a technique to exploit task-level parallelism to improve instruction- and data-level parallelism,” in Proc. IEEE International Workshop on Power And Timing Modeling, Optimization and Simulation (PATMOS), pp.12–23, Springer Verlag LNCS, Sept. 2006.

[61] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, “Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling,” in Proc. Design Automation and Test in Europe (DATE), pp.296–301, March 2003.

[62] L. Benini and G. De Micheli, “System-level power optimization: Techniques and tools,” ACM Trans. on Design Automation of Electronic Systems, vol.5, no.2, pp.115–192, April 2000.

Appendix A

BNF of processor specification description

This chapter shows BNF of the processor specification description described in Chapter 3.

```
<InstructionSetSpec> ::= 'mod' <modName> '{'
                         <DeclarationPart>
                         <BehaviorPart>
                         '}'
<modName> ::= <identifier>

<DeclarationPart> ::= <OperationWidthDecl> <SlotDecl> <PipelineStageDecl>
                      <DesignPriorityDecl> <ResourceDeclPart>
                      <ResourceGroupDeclPart> <OperationTypeDeclPart>
                      <OperationDeclPart> <OperationGroupDeclPart> <IODecl> <InterruptDecl>


<OperationWidthDecl> ::= 'operation_length' <OperationWidth> ';'
<OperationWidth> ::= <NaturalNumber>

<SlotDecl> ::= 'slots' '{' <SlotNames> '}' ';'
<SlotNames> ::= <SlotName> {  ',' <SlotName> }
<SlotName> ::= <Identifier>

<PipelineStageDecl> ::= <StageDecls> <BuildStageDef> <DispatchStageDef>
                        <DecodeStageDef> <RegisterBypassDef> <MemoryBypassDef>
                        <DelaySlotDef> <DelaySlotNum>
<StageDecls> ::= 'stages' '{' <StageName> {  ',' <StageName>  } ' } ' ';'
<StageName> ::= <Identifier>
<BuildStageDef> ::= 'build_stage' <StageName> ';'
<DispatchStageDef> ::= 'dispatch_stage' <StageName> ';'
<DecodeStageDef> ::= 'decode_stage' <StageName> ';'
<RegisterBypassDef> ::= 'use_register_bypass <YorN> ';'
<MemoryBypassDef> ::= 'use_memory_bypass <YorN> ';'
<DelaySlotDef> ::= 'use_delayed_branch' <YorN> ';'
<DelaySlotNum> ::= 'num_delayed_slots' <NaturalNumber> ';'
<YorN> ::= 'yes' |'no'

<DesignPriorityDecl> ::= 'priority' <DesignPriority> ';'
<DesignPriority> ::= <String>

<ResourceDeclPart> ::= {  <ResourceDecl>  }
<ResourceDecl> ::= 'resource' <ResourceName> [  ':'  <ResourceUsageDef>  ] '{'
                   'model' <FHMModelName> ';'
                   'for_simulation' '{' <FHMParameterDef> '}' ';'
                   'for_synthesis' '{' <FHMParameterDef> '}' ';'
                   '}' ';'
```

```
<ResourceName> ::= <Identifier>
<ResourceUsageDef> ::= 'program_counter' |'instr_memory' |'data_memory' |
                       'register_file' |'status_register' |'fetch_register' |
                       'instr_register' |'flag_register' |'plain_register'
<FHMParameterDef> ::= 'design_level' <FHMDesignLevel> ';'
                      'parameter' <FHMParameter> ';'
<FHMModelName> ::= <String>
<FHMDesignLevel> ::= <String>
<FHMParameter> ::= <String>

<ResourceGroupDeclPart> ::= {  <ResourceGroupDecl>  }
<ResourceGroupDecl> ::= 'resgroup' <ResourceGroupName> '{'
                        [  'member' ':'  <ResourceNames> ';'  ]
                        [  'read_access' ':'  <ResourceNames> ';'  ]
                        [  'write_access' ':'  <ResourceNames> ';'  ]
                        '}' ';'
<ResourceNames> ::= <ResourceName> {  ',' <ResourceName>  }
<ResourceGroupName> ::= <Identifier>


<OperationTypeDeclPart> ::= <OperationTypeDecl>
                            {  <OperationTypeDecl>  }
<OperationTypeDecl> ::= 'opetype' <OperationTypeName> '{'
                        {  <FieldDef>  }
                        '}' ';'
<OperationTypeName> ::= <Identifier>
<FieldDef> ::= <VLIWInstDelimiterFieldDef> |
               <OperandFieldDef> |<OpecodeFieldDef> |<ReservedFieldDef>
<VLIWInstDelimiterFieldDef> ::= 'terminate_flag' <BitRange>
                                <FieldName> ';'
<OperandFieldDef> ::= 'operand' <BitRange> <FieldName> ';'
<OpecodeFieldDef> ::= 'opecode' <BitRange> <FieldName>
                      [ '=' <BinaryConstant> ] ';'
<ReservedFieldDef> ::= 'reserved' <BitRange> <FieldName> ';'
<FieldName> ::= <Identifier>


<OperationDeclPart> ::= {  <OperationDecl>  }
<OperationDecl> ::= 'operation' <OperationName> ':'
                    <OperationTypeName> '{'
                    {  <OpecodeDef>  } <Format>'}' ';'
<OperationName> ::= <Identifier>
<OpecodeDef> ::= ('opecode' |'reserved') <FieldName> '=' <BinaryConstant> ';'
<Format> ::= 'format' '{' <ElementList> '}' ';'
<ElementList> ::= <Element> { ',' <Operand> }
<Element> ::= <Identifier> |' '' ' <Identifier> ' '' '

<OperationGroupDeclPart> ::= {  <OperationGroupDecl>  }
<OperationGroupDecl> ::= 'opegroup' <OperationGroupName> '{'
                         <OperationName> {  ',' <OperationName>  }
                         '}' ';'
<OperationGroupName> ::= <Identifier>


<IODecl> ::= <TopModuleNameDef>
             <ClockPortDef> <ResetPortDef> <UserPortDefs>

<TopModuleNameDef> ::= 'top_module' <TopModuleName> ';'
<TopModuleName> ::= <Identifier>
<ClockPortDef> ::= 'clock_port' <PortName> ';'
<ResetPortDef> ::= 'reset_port' <PortName> ';'
<UserPortDefs> ::= <UserPortDef> {  <UserPortDef>  }
<UserPortDef> ::= 'port' [ <BitRangeDef> ] <PortName> '{'
                  'direction' ('in' |'out' |'inout') ';'
                  'connect_to' <Destination> ';'
                  '}' ';'
<Destination> ::= 'internal_controller' |<ResourceName> '.'  <PortName>
<PortName> ::= <Identifier>
```

```
<InterruptDeclPart> ::= <ResetInterruptDecl>
                        [  <NMIDecl>  ]
                        [  <ExternalInterruptDecl>  ]
                        {  <InternalInterruptDecl>  }
<ResetInterruptDecl> ::= 'reset_interrupt' <InterruptName> '{'
                         <InterruptCauseCondition>
                         '}' ';'
<NMIDecl> ::= 'nonmaskable_interrupt' <InterruptName> '{'
              <InterruptCauseCondition>
              '}' ';'
<ExternalInterruptDecl> ::= 'external_interrupt' <InterruptName> '{'
                            <InterruptCauseCondition>
                            [ <InterruptMaskCondition> ]
                            '}' ';'
<InternalInterruptDecl> ::= 'internal_interrupt' <InterruptName> '{'
                            'cause_condition_type' ('decode_error' |'instr_specific') ';'
                            [ <InterruptMaskCondition> ]
                            '}' ';'
<InterruptCauseCondition> ::= 'cause_condition' '{'
                              'port' <PortName> ';'
                              'active_value' <BitLiteral> ';'
                              '}' ';'
<InterruptMaskCondition> ::= 'mask_condition' '{'
                             'mask_register' <ResourceName> ';'
                             'mask_bitpos' <NonNegativeInteger> ';'
                             'active_value' <BitLiteral> ';'
                             '}' ';'
<InterruptName> ::= <Identifier>

<BehaviorPart> ::= <InterruptDefPart>
                   <CommonStageDef>
                   <DispatchTable>
                   <OperationDefPart>


<InterruptDefPart> ::= {  <InterruptDef>  }
<InterruptDef> ::= 'catch_interrupt' <InterruptName> '{'
                   <VariableDeclPart>
                   <InterruptDefExpressions>
                   '}' ';'
<InterruptDefExpressions> ::= {  <InterruptDefExpression>  }
<InterruptDefExpression> ::= <Assignment> |
                             <ConditionalAssignment>

<CommonStageDef> ::= 'common_pre_dispatch' '{'
                     <CommonStageDesc>
                     '}' ';'
<CommonStageDesc> ::= <VariableDeclPart>
                      <MicroOperationDescriptionPart>
<VariableDeclPart> ::= {  <VariableDecl>  }
<VariableDecl> ::= <WireDecl>
<WireDecl> ::= 'wire' [  <BitRange>  ] <WireName> ';'
<WireName> ::= <Identifier>

<DispatchTable> ::= 'dispatch_table' '{'
                    <S-OGTable>
                    <S-RGTable>
                    <OG-RGTable>
                    '}' ';'
<S-OGTable> ::= 'slot_opegroup' '{'
                {  <S-OGRelation>  }
                '}' ';'
<S-OGRelation> ::= '{' <S-OGPair> {  ',' <S-OGPair>  } '}' ';'
<S-OGPair> ::= <SlotName> ':'  <OGDef>
<OGDef> ::= <OperationGroupName> |'null'
<S-RGTable> ::= 'slot_resgroup' '{'
                {  <S-RGRelation>  }
```

```
               '}' ';'
<S-RGRelation> ::= <SlotName> ':'  <RGs> ';'
<RGs> ::= <ResourceGroupName> {  ',' <ResourceGroupName>  }
<OG-RGTable> ::= 'opegroup_resgroup' '{'
                 {  <OG-RGRelation>  }
                 '}' ';'
<OG-RGRelation> ::= <OperationGroupName> ':'
                    <ResourceGroupName> {  ',' <ResourceGroupName>  } ';'

<OperationDefPart> ::= {  <OperationDef>  }
<OperationDef> ::= 'micro_operation' <OperationName> 'on'
                   <ResourceGroupName> '{'
                   <OperationBehaviorDesc>
                   '}' ';'
<OperationBehaviorDesc> ::= <GlobalVariableDeclPart>
                            <VariableDeclPart>
                            <MicroOperationDescriptionPart>
<GlobalVariableDeclPart> ::= {  <GlobalVariableDecl>  }
<GlobalVariableDecl> ::= 'extern' <VariableDecl>

<MicroOperationDescriptionPart> ::= {  <MicroOperationDescription>  }
<MicroOperationDescription> ::= 'stage' <StageNumber> ':'  '{'
                                <StageVariableDeclPart>
                                <Expressions>
                                '}' ';'
<StageNumber> ::= <NaturalNumber>
<StageVariableDeclPart> ::= <VariableDeclPart>
<Expressions> ::= {  <Expression>  }
<Expression> ::= <Assignment> |<ConditionalAssignment> |
                 <ConditionalFunctionalExecution> |<InternalInterrupt> |
                 <ConditionalInternalInterrupt>
<Assignment> ::= <LeftSide> '=' <RightSide> ';'
<LeftSide> ::= <VariableName> |<VariableNameSet> |'null'
<RightSide> ::= <BitwiseAND> |<BitwiseOR> |<BitwiseNOT> |<Comparison> |
                <Aggregation> |<ResourceRef> |<BitSelect> |<RangeSelect> |
                <BinaryConstant> |
                <VariableRef>
<BitwiseAND> ::= <VariableRef> '&' <VariableRef>
<BitwiseOR> ::= <VariableRef> '|' <VariableRef>
<BitwiseNOT> ::= '~' <VariableRef>
<Comparison> ::= <VariableRef> <RationalOperator> <BinaryConstant>
<RationalOperator> ::= '==' |'!='
<VariableRef> ::= <VariableName> |<FieldName>
<VariableName> ::= <WireName>
<VariableNameSet> ::= '<' <VariableName> {  ',' <VariableName>  } '>'
<Aggregation> ::= '<' <VariableRef> {  ',' <VariableRef>  } '>'
<ResourceRef> ::= <ResourceName> '.'  <FunctionName> '(' [  <Parameters>  ] ')'
<FunctionName> ::= <Identifier>
<Parameters> ::= <Parameter> {  ',' <Parameter>  }
<Parameter> ::= <VariableRef>
<BitSelect> ::= <VariableRef> '[' <NonNegativeInteger> ']'
<RangeSelect> ::= <VariableRef> <BitRange>
<ConditionalAssignment> ::= <LeftSide> '=' '(' <BitVariableRef> ')' '?'
                <VariableRef> ':'  <VariableRef> ';'
<BitVariableRef> ::= <VariableRef>
<ConditionalFunctionalExecution> ::= <LeftSide> '=' '[' <BitVariableRef> ']' '?'
                <ResourceFunctionDef> ';'
<ResourceFunctionDef> ::= <ResourceName> '.'  <FunctionName> '(' [  <Parameters>  ] ')'
<InternalInterrupt> ::= 'throw' <InterruptName> ';'
<ConditionalInternalInterrupt> ::= '[' <BitVariableRef> ']' 'throw' <InterruptName> ';'
```

Appendix B

Processor description for the proposed VLIW generation method

This chapter shows a sample of processor specification description of VLIW processor shown in Section 3.7.2 that is designed based on the proposed resource group assignment algorithm. Since it is too long (11664 lines), this chapter only shows an excerpt from it.

```
mod CPU {
  operation_length 32;
  slots { slot1, slot2, slot3, slot4 };
  stages { IF, ID, EXE, MEM, WB };
  build_stage IF;
  dispatch_stage ID;
  decode_stage ID;
  use_register_bypass no;
  use_memory_bypass no;
   use_delayed_branch yes;
   num_delayed_slots 1;
   priority "Area";
   resource PC : program_counter {
     model "/workdb/peas/pcu";
     for_simulation {
       design_level "Behavior";
       parameter "bit_width=32 increment_step=8 adder_algorithm=cla";
     };
     for_synthesis {
       design_level "Synthesis";
       parameter "bit_width=32 increment_step=8 adder_algorithm=cla";
     };
   };
   resource IR : instr_register {
     model "/basicfhmdb/storage/register";
     for_simulation {
       design_level "Behavior";
       parameter "bit_width=128";
     };
     for_synthesis {
       design_level "Synthesis";
       parameter "bit_width=128";
     };
   };
   resource IMAU : instr_memory {
     model "/workdb/peas/imau";
     for_simulation {
       design_level "Behavior";
       parameter "bit_width=128 address_space=32";
     };
```

```
41      for_synthesis {
42        design_level "Synthesis";
43        parameter "bit_width=128 address_space=32";
44      };
45    };
46    resource DMAU : data_memory {
47      model "/workdb/peas/dmau";
48      for_simulation {
49        design_level "Behavior";
50        parameter "bit_width=32 address_space=32 access_width=8";
51      };
52      for_synthesis {
53        design_level "Synthesis";
54        parameter "bit_width=32 address_space=32 access_width=8";
55      };
56    };
57    resource GPR : register_file {
58      model "/basicfhmdb/storage/registerfile";
59      for_simulation {
60        design_level "Behavior";
61        parameter "bit_width=32 num_register=32 num_read_port=8 num_write_port=4";
62      };
63      for_synthesis {
64        design_level "Synthesis";
65        parameter "bit_width=32 num_register=32 num_read_port=8 num_write_port=4";
66      };
67    };
68    resource ALU0 {
69      model "/basicfhmdb/computational/alu";
70      for_simulation {
71        design_level "Behavior";
72        parameter "bit_width=32 algorithm=cla";
73      };
74      for_synthesis {
75        design_level "Synthesis";
76        parameter "bit_width=32 algorithm=cla";
77      };
78    };
79    resource EXT00 {
80      model "/basicfhmdb/computational/extender";
81      for_simulation {
82        design_level "Behavior";
83        parameter "bit_width=16 bit_width_out=32";
84      };
85      for_synthesis {
86        design_level "Synthesis";
87        parameter "bit_width=16 bit_width_out=32";
88      };
89    };
90    resource ALU1 {
91      model "/basicfhmdb/computational/alu";
92      for_simulation {
93        design_level "Behavior";
94        parameter "bit_width=32 algorithm=cla";
95      };
96      for_synthesis {
97        design_level "Synthesis";
98        parameter "bit_width=32 algorithm=cla";
99      };
100   };
101   resource EXT01 {
102     model "/basicfhmdb/computational/extender";
103     for_simulation {
104       design_level "Behavior";
105       parameter "bit_width=16 bit_width_out=32";
106     };
107     for_synthesis {
108       design_level "Synthesis";
```

```
109       parameter "bit_width=16 bit_width_out=32";
110     };
111   };
112   resource ALU2 {
113     model "/basicfhmdb/computational/alu";
114     for_simulation {
115       design_level "Behavior";
116       parameter "bit_width=32 algorithm=cla";
117     };
118     for_synthesis {
119       design_level "Synthesis";
120       parameter "bit_width=32 algorithm=cla";
121     };
122   };
123   resource EXT02 {
124     model "/basicfhmdb/computational/extender";
125     for_simulation {
126       design_level "Behavior";
127       parameter "bit_width=16 bit_width_out=32";
128     };
129     for_synthesis {
130       design_level "Synthesis";
131       parameter "bit_width=16 bit_width_out=32";
132     };
133   };
134   resource MUL0 {
135     model "/basicfhmdb/computational/multiplier";
136     for_simulation {
137       design_level "Behavior";
138       parameter "bit_width=32 algorithm=seq adder_algorithm=cla data_type=two_complement";
139     };
140     for_synthesis {
141       design_level "Synthesis";
142       parameter "bit_width=32 algorithm=seq adder_algorithm=cla data_type=two_complement";
143     };
144   };
145   resource MUL1 {
146     model "/basicfhmdb/computational/multiplier";
147     for_simulation {
148       design_level "Behavior";
149       parameter "bit_width=32 algorithm=seq adder_algorithm=cla data_type=two_complement";
150     };
151     for_synthesis {
152       design_level "Synthesis";
153       parameter "bit_width=32 algorithm=seq adder_algorithm=cla data_type=two_complement";
154     };
155   };
156   resource DIV0 {
157     model "/basicfhmdb/computational/divider";
158     for_simulation {
159       design_level "Behavior";
160       parameter "bit_width=32 algorithm=seq adder_algorithm=cla data_type=two_complement";
161     };
162     for_synthesis {
163       design_level "Synthesis";
164       parameter "bit_width=32 algorithm=seq adder_algorithm=cla data_type=two_complement";
165     };
166   };
167   resource DIV1 {
168     model "/basicfhmdb/computational/divider";
169     for_simulation {
170       design_level "Behavior";
171       parameter "bit_width=32 algorithm=seq adder_algorithm=cla data_type=two_complement";
172     };
173     for_synthesis {
174       design_level "Synthesis";
175       parameter "bit_width=32 algorithm=seq adder_algorithm=cla data_type=two_complement";
176     };
```

```
177   };
178   resource SFT0 {
179     model "/basicfhmdb/computational/shifter";
180     for_simulation {
181       design_level "Behavior";
182       parameter "bit_width=32 amount=variable";
183     };
184     for_synthesis {
185       design_level "Synthesis";
186       parameter "bit_width=32 amount=variable";
187     };
188   };
189   resource SFT1 {
190     model "/basicfhmdb/computational/shifter";
191     for_simulation {
192       design_level "Behavior";
193       parameter "bit_width=32 amount=variable";
194     };
195     for_synthesis {
196       design_level "Synthesis";
197       parameter "bit_width=32 amount=variable";
198     };
199   };
200   resource SFT2 {
201     model "/basicfhmdb/computational/shifter";
202     for_simulation {
203       design_level "Behavior";
204       parameter "bit_width=32 amount=variable";
205     };
206     for_synthesis {
207       design_level "Synthesis";
208       parameter "bit_width=32 amount=variable";
209     };
210   };
211   resource EXT1 {
212     model "/basicfhmdb/computational/extender";
213     for_simulation {
214       design_level "Behavior";
215       parameter "bit_width=28 bit_width_out=32";
216     };
217     for_synthesis {
218       design_level "Synthesis";
219       parameter "bit_width=28 bit_width_out=32";
220     };
221   };
222
223   resgroup RG01ASFT {
224     member : SFT0;
225     read_access : GPR.data_out0, GPR.data_out1;
226     write_access : GPR.r_sel0, GPR.r_sel1, GPR.data_in0, GPR.w_sel0, GPR.w_enb0;
227   };
228   resgroup RG02ASFT {
229     member : SFT0;
230     read_access : GPR.data_out2, GPR.data_out3;
231     write_access : GPR.r_sel2, GPR.r_sel3, GPR.data_in1, GPR.w_sel1, GPR.w_enb1;
232   };
233   resgroup RG02BSFT {
234     member : SFT1;
235     read_access : GPR.data_out2, GPR.data_out3;
236     write_access : GPR.r_sel2, GPR.r_sel3, GPR.data_in1, GPR.w_sel1, GPR.w_enb1;
237   };
238   resgroup RG03BSFT {
239     member : SFT1;
240     read_access : GPR.data_out4, GPR.data_out5;
241     write_access : GPR.r_sel4, GPR.r_sel5, GPR.data_in2, GPR.w_sel2, GPR.w_enb2;
242   };
243   resgroup RG03CSFT {
244     member : SFT2;
```

```
245     read_access : GPR.data_out4, GPR.data_out5;
246     write_access : GPR.r_sel4, GPR.r_sel5, GPR.data_in2, GPR.w_sel2, GPR.w_enb2;
247   };
248   resgroup RG04CSFT {
249     member : SFT2;
250     read_access : GPR.data_out6, GPR.data_out7;
251     write_access : GPR.r_sel6, GPR.r_sel7, GPR.data_in3, GPR.w_sel3, GPR.w_enb3;
252   };
253   resgroup RG01AALU {
254     member : ALU0, EXT00;
255     read_access : GPR.data_out0, GPR.data_out1;
256     write_access : GPR.r_sel0, GPR.r_sel1, GPR.data_in0, GPR.w_sel0, GPR.w_enb0;
257   };
258   resgroup RG02AALU {
259     member : ALU0, EXT00;
260     read_access : GPR.data_out2, GPR.data_out3;
261     write_access : GPR.r_sel2, GPR.r_sel3, GPR.data_in1, GPR.w_sel1, GPR.w_enb1;
262   };
263   resgroup RG02BALU {
264     member : ALU1, EXT01;
265     read_access : GPR.data_out2, GPR.data_out3;
266     write_access : GPR.r_sel2, GPR.r_sel3, GPR.data_in1, GPR.w_sel1, GPR.w_enb1;
267   };
268   -------- snip --------
269   resgroup RG03CMEM {
270     member : ALU2, EXT02, DMAU;
271     read_access : GPR.data_out4, GPR.data_out5;
272     write_access : GPR.r_sel4, GPR.r_sel5, GPR.data_in2, GPR.w_sel2, GPR.w_enb2;
273   };
274   resgroup RG04CMEM {
275     member : ALU2, EXT02, DMAU;
276     read_access : GPR.data_out6, GPR.data_out7;
277     write_access : GPR.r_sel6, GPR.r_sel7, GPR.data_in3, GPR.w_sel3, GPR.w_enb3;
278   };
279   resgroup RG01ANOP {
280     read_access : GPR.data_out0, GPR.data_out1;
281     write_access : GPR.r_sel0, GPR.r_sel1, GPR.data_in0, GPR.w_sel0, GPR.w_enb0;
282   };
283   resgroup RG02ANOP {
284     read_access : GPR.data_out2, GPR.data_out3;
285     write_access : GPR.r_sel2, GPR.r_sel3, GPR.data_in1, GPR.w_sel1, GPR.w_enb1;
286   };
287   resgroup RG03ANOP {
288     read_access : GPR.data_out4, GPR.data_out5;
289     write_access : GPR.r_sel4, GPR.r_sel5, GPR.data_in2, GPR.w_sel2, GPR.w_enb2;
290   };
291   resgroup RG04ANOP {
292     read_access : GPR.data_out6, GPR.data_out7;
293     write_access : GPR.r_sel6, GPR.r_sel7, GPR.data_in3, GPR.w_sel3, GPR.w_enb3;
294   };
```

```
449  opetype R_R{
450    opecode [31:26] opecode__;
451    operand [25:21] rs0;
452    operand [20:16] rs1;
453    operand [15:11] rd;
454    opecode [10:0] func;
455  };
456  opetype R_I{
457    opecode [31:26] opecode__;
458    operand [25:21] rs0;
459    operand [20:16] rd;
460    operand [15:0] const__;
461  };
462  opetype L_S{
463    opecode [31:26] opecode__;
464    operand [25:21] rs0;
465    operand [20:16] rd;
466    operand [15:0] const__;
467  };
468  opetype B{
469    opecode [31:26] opecode__;
470    operand [25:21] rs0;
471    opecode [20:16] fld_20_16;
472    operand [15:0] const__;
473  };
474  opetype J{
475    opecode [31:26] opecode__;
476    operand [25:0] const__;
477  };
478  opetype JR{
479    opecode [31:26] fld_31_26;
480    operand [25:21] rs0;
481    opecode [20:11] fld_20_11;
482    opecode [10:0] func;
483  };
484  opetype LHI{
485    opecode [31:26] opecode__;
486    opecode [25:21] fld_25_21;
487    operand [20:16] rd;
488    operand [15:0] const__;
489  };
490  operation ADD : R_R{
491    opecode opecode__ = "000000";
492    opecode func = "00000100000";
493  };
494  operation ADDU : R_R{
495    opecode opecode__ = "000000";
496    opecode func = "00000100001";
497  };
498  operation ADDI : R_I{
499    opecode opecode__ = "001000";
500  };
501  operation ADDUI : R_I{
502    opecode opecode__ = "001001";
503  };
504  operation SUB : R_R{
505    opecode opecode__ = "000000";
506    opecode func = "00000100010";
507  };
508  operation SUBU : R_R{
509    opecode opecode__ = "000000";
510    opecode func = "00000100011";
511  };
512  operation SUBI : R_I{
513    opecode opecode__ = "001010";
514  };
515  operation SUBUI : R_I{
516    opecode opecode__ = "001011";
517  };
518  operation MULT : R_R{
519    opecode opecode__ = "000000";
520    opecode func = "00000011000";
521  };
522  operation MULTU : R_R{
523    opecode opecode__ = "000000";
524    opecode func = "00000011001";
525  };
526  operation DIV : R_R{
527    opecode opecode__ = "000000";
528    opecode func = "00000011010";
529  };
530  operation DIVU : R_R{
531    opecode opecode__ = "000000";
532    opecode func = "00000011011";
533  };
534  operation AND : R_R{
535    opecode opecode__ = "000000";
536    opecode func = "00000100100";
537  };
538  operation ANDI : R_I{
539    opecode opecode__ = "001100";
540  };
541  operation OR : R_R{
542    opecode opecode__ = "000000";
543    opecode func = "00000100101";
544  };
545  operation ORI : R_I{
546    opecode opecode__ = "001101";
547  };
548  operation XOR : R_R{
549    opecode opecode__ = "000000";
550    opecode func = "00000100110";
551  };
552  operation XORI : R_I{
553    opecode opecode__ = "001110";
554  };
555  operation SLL : R_R{
556    opecode opecode__ = "000000";
557    opecode func = "00000000000";
558  };
559  operation SRL : R_R{
560    opecode opecode__ = "000000";
561    opecode func = "00000000010";
562  };
563  operation SRA : R_R{
564    opecode opecode__ = "000000";
565    opecode func = "00000000011";
566  };
567  operation SLLI : R_I{
568    opecode opecode__ = "010000";
569  };
570  operation SRLI : R_I{
571    opecode opecode__ = "010001";
572  };
573  operation SRAI : R_I{
574    opecode opecode__ = "010010";
575  };
576  operation SLT : R_R{
577    opecode opecode__ = "000000";
578    opecode func = "00000101010";
579  };
580  operation SGT : R_R{
581    opecode opecode__ = "000000";
582    opecode func = "00000101011";
```

```
583   };
584   operation SLE : R_R{
585     opecode opecode__ = "000000";
586     opecode func = "00000101100";
587   };
588   operation SGE : R_R{
589     opecode opecode__ = "000000";
590     opecode func = "00000101101";
591   };
592   operation SEQ : R_R{
593     opecode opecode__ = "000000";
594     opecode func = "00000101110";
595   };
596   operation SNE : R_R{
597     opecode opecode__ = "000000";
598     opecode func = "00000101111";
599   };
600   operation SLTI : R_I{
601     opecode opecode__ = "011010";
602   };
603   operation SGTI : R_I{
604     opecode opecode__ = "011011";
605   };
606   operation SLEI : R_I{
607     opecode opecode__ = "011100";
608   };
609   operation SGEI : R_I{
610     opecode opecode__ = "011101";
611   };
612   operation SEQI : R_I{
613     opecode opecode__ = "011110";
614   };
615   operation SNEI : R_I{
616     opecode opecode__ = "011111";
617   };
618   operation LHI : LHI{
619     opecode opecode__ = "001111";
620     opecode fld_25_21 = "00000";
621   };
622   operation LB : L_S{
623     opecode opecode__ = "100000";
624   };
625   operation LH : L_S{
626     opecode opecode__ = "100001";
627   };
628   operation LW : L_S{
629     opecode opecode__ = "100011";
630   };
631   -------- snip --------
632   operation SGTU : R_R{
633     opecode opecode__ = "000000";
634     opecode func = "00000111011";
635   };
636   operation SLEU : R_R{
637     opecode opecode__ = "000000";
638     opecode func = "00000111100";
639   };
640   operation SGEU : R_R{
641     opecode opecode__ = "000000";
642     opecode func = "00000111101";
643   };
644   opegroup OG_SFT {
645     SLL, SRL, SRA, SLLI, SRLI, SRAI
646   };
647   opegroup OG_ALU {
648     ADD, ADDU, ADDI, ADDUI, SUB, SUBU,
649     SUBI, SUBUI, AND, ANDI, OR, ORI,
650     XOR, XORI, SLT, SGT, SLE, SGE,
651     SEQ, SNE, SLTI, SGTI, SLEI, SGEI,
652     SEQI, SNEI, SLTU, SGTU, SLEU, SGEU
653   };
654   opegroup OG_JMP {
655     BEQZ, BNEZ, J, JAL, JR, JALR
656   };
657   opegroup OG_MEM {
658     LB, LH, LW, LBU, LHU, SB, SH, SW
659   };
660   opegroup OG_MUL {
661     MULT, MULTU
662   };
663   opegroup OG_DIV {
664     DIV, DIVU, MOD, MODU
665   };
666   opegroup OG_NOP {
667     LHI
668   };
669   top_module CPU;
670   clock_port CLK;
671   reset_port Reset;
672   port [31:0] InstAB {
673     direction out;
674     connect_to IMAU.addr_bus;
675   };
676   port [127:0] InstDB {
677     direction in;
678     connect_to IMAU.data_bus;
679   };
680   port [31:0] DataAB {
681     direction out;
682     connect_to DMAU.addr_bus;
683   };
684   port [31:0] DataDB {
685     direction inout;
686     connect_to DMAU.data_bus;
687   };
688   port DataReq {
689     direction out;
690     connect_to DMAU.req_bus;
691   };
692   port DataAck {
693     direction in;
694     connect_to DMAU.ack_bus;
695   };
696   port [3:0] DataWm {
697     direction out;
698     connect_to DMAU.w_mode_bus;
699   };
700   reset_interrupt reset{
701     cause_condition {
702       port Reset;
703       active_value '1';
704     };
705   };
706   common_pre_dispatch {
707     stage 1 {
708       wire [31:0] current_pc;
709       wire [127:0] inst;
710
711       current_pc = PC.read();
712       inst = IMAU.read(current_pc);
713       null = IR.write(inst);
714       null = PC.inc(); };
715     stage 2 {};
716   };
```

```
dispatch_table {
  slot_opegroup {
    {slot1: OG_SFT, slot2: OG_SFT, slot3: OG_SFT, slot4: OG_JMP};
    {slot1: OG_SFT, slot2: OG_SFT, slot3: OG_SFT, slot4: OG_ALU};
    {slot1: OG_SFT, slot2: OG_SFT, slot3: OG_SFT, slot4: OG_MUL};
    {slot1: OG_SFT, slot2: OG_SFT, slot3: OG_SFT, slot4: OG_DIV};
    {slot1: OG_SFT, slot2: OG_SFT, slot3: OG_SFT, slot4: OG_MEM};
    {slot1: OG_SFT, slot2: OG_SFT, slot3: OG_SFT, slot4: OG_NOP};
    {slot1: OG_SFT, slot2: OG_SFT, slot3: OG_JMP, slot4: OG_SFT};
    {slot1: OG_SFT, slot2: OG_SFT, slot3: OG_JMP, slot4: OG_ALU};
    {slot1: OG_SFT, slot2: OG_SFT, slot3: OG_JMP, slot4: OG_MUL};
    {slot1: OG_SFT, slot2: OG_SFT, slot3: OG_JMP, slot4: OG_DIV};
    {slot1: OG_SFT, slot2: OG_SFT, slot3: OG_JMP, slot4: OG_NOP};
    {slot1: OG_SFT, slot2: OG_SFT, slot3: OG_ALU, slot4: OG_SFT};
    {slot1: OG_SFT, slot2: OG_SFT, slot3: OG_ALU, slot4: OG_JMP};
    {slot1: OG_SFT, slot2: OG_SFT, slot3: OG_ALU, slot4: OG_ALU};
    {slot1: OG_SFT, slot2: OG_SFT, slot3: OG_ALU, slot4: OG_MUL};
    {slot1: OG_SFT, slot2: OG_SFT, slot3: OG_ALU, slot4: OG_DIV};
    {slot1: OG_SFT, slot2: OG_SFT, slot3: OG_ALU, slot4: OG_MEM};
    {slot1: OG_SFT, slot2: OG_SFT, slot3: OG_ALU, slot4: OG_NOP};
    {slot1: OG_SFT, slot2: OG_SFT, slot3: OG_MUL, slot4: OG_SFT};
    {slot1: OG_SFT, slot2: OG_SFT, slot3: OG_MUL, slot4: OG_JMP};
    {slot1: OG_SFT, slot2: OG_SFT, slot3: OG_MUL, slot4: OG_ALU};
    {slot1: OG_SFT, slot2: OG_SFT, slot3: OG_MUL, slot4: OG_MUL};
    {slot1: OG_SFT, slot2: OG_SFT, slot3: OG_MUL, slot4: OG_DIV};
    {slot1: OG_SFT, slot2: OG_SFT, slot3: OG_MUL, slot4: OG_MEM};
    {slot1: OG_SFT, slot2: OG_SFT, slot3: OG_MUL, slot4: OG_NOP};
    {slot1: OG_SFT, slot2: OG_SFT, slot3: OG_DIV, slot4: OG_SFT};
    {slot1: OG_SFT, slot2: OG_SFT, slot3: OG_DIV, slot4: OG_JMP};
    {slot1: OG_SFT, slot2: OG_SFT, slot3: OG_DIV, slot4: OG_ALU};
    {slot1: OG_SFT, slot2: OG_SFT, slot3: OG_DIV, slot4: OG_MUL};
    {slot1: OG_SFT, slot2: OG_SFT, slot3: OG_DIV, slot4: OG_DIV};
    {slot1: OG_SFT, slot2: OG_SFT, slot3: OG_DIV, slot4: OG_MEM};
    {slot1: OG_SFT, slot2: OG_SFT, slot3: OG_DIV, slot4: OG_NOP};
    {slot1: OG_SFT, slot2: OG_SFT, slot3: OG_MEM, slot4: OG_SFT};
    -------- snip (a total of 1565 patterns) --------
    {slot1: OG_NOP, slot2: OG_NOP, slot3: OG_NOP, slot4: OG_SFT};
    {slot1: OG_NOP, slot2: OG_NOP, slot3: OG_NOP, slot4: OG_JMP};
    {slot1: OG_NOP, slot2: OG_NOP, slot3: OG_NOP, slot4: OG_ALU};
    {slot1: OG_NOP, slot2: OG_NOP, slot3: OG_NOP, slot4: OG_MUL};
    {slot1: OG_NOP, slot2: OG_NOP, slot3: OG_NOP, slot4: OG_DIV};
    {slot1: OG_NOP, slot2: OG_NOP, slot3: OG_NOP, slot4: OG_MEM};
    {slot1: OG_NOP, slot2: OG_NOP, slot3: OG_NOP, slot4: OG_NOP};
  };
  slot_resgroup {
    slot1: RG01ASFT, RG01AJMP, RG01AALU, RG01AMUL, RG01ADIV, RG01AMEM, RG01ANOP;
    slot2: RG02ASFT, RG02BSFT, RG02AJMP, RG02BJMP, RG02AALU, RG02BALU,
           RG02AMUL, RG02BMUL, RG02ADIV, RG02BDIV, RG02AMEM, RG02BMEM, RG02ANOP;
    slot3: RG03BSFT, RG03CSFT, RG03AJMP, RG03BJMP, RG03CJMP, RG03BALU,
           RG03CALU, RG03AMUL, RG03BMUL, RG03ADIV, RG03BDIV, RG03AMEM,
           RG03BMEM, RG03CMEM, RG03ANOP;
    slot4: RG04CSFT, RG04AJMP, RG04BJMP, RG04CJMP, RG04CALU, RG04BMUL,
           RG04BDIV, RG04AMEM, RG04BMEM, RG04CMEM, RG04ANOP;
  };
  opegroup_resgroup {
    OG_SFT: RG01ASFT, RG02ASFT, RG02BSFT, RG03BSFT, RG03CSFT, RG04CSFT;
    OG_JMP: RG01AJMP, RG02AJMP, RG02BJMP, RG03AJMP, RG03BJMP, RG03CJMP,
            RG04AJMP, RG04BJMP, RG04CJMP;
    OG_ALU: RG01AALU, RG02AALU, RG02BALU, RG03BALU, RG03CALU, RG04CALU;
    OG_MUL: RG01AMUL, RG02AMUL, RG02BMUL, RG03AMUL, RG03BMUL, RG04BMUL;
    OG_DIV: RG01ADIV, RG02ADIV, RG02BDIV, RG03ADIV, RG03BDIV, RG04BDIV;
    OG_MEM: RG01AMEM, RG02AMEM, RG02BMEM, RG03AMEM, RG03BMEM, RG03CMEM,
            RG04AMEM, RG04BMEM, RG04CMEM;
    OG_NOP: RG01ANOP, RG02ANOP, RG03ANOP, RG04ANOP;
  };
};
```

```
micro_operation SLL on RG01ASFT {
    wire [31:0] source0;
    wire [31:0] source1;
    wire [31:0] result;
  stage 2 {
    source0 = GPR.read0(rs0);
    source1 = GPR.read1(rs1);
  };
  stage 3 {
    wire [4:0] shamt;

    shamt = source1[4:0];
    result = SFT0.sll(source0, shamt);
  };
  stage 4 {
  };
  stage 5 {
    null = GPR.write0(rd, result);
  };
};
micro_operation SRL on RG01ASFT {
    wire [31:0] source0;
    wire [31:0] source1;
    wire [31:0] result;
  stage 2 {
    source0 = GPR.read0(rs0);
    source1 = GPR.read1(rs1);
  };
  stage 3 {
    wire [4:0] shamt;

    shamt = source1[4:0];
    result = SFT0.srl(source0, shamt);
  };
  stage 4 {
  };
  stage 5 {
    null = GPR.write0(rd, result);
  };
};
-------- snip --------
micro_operation SRAI on RG01ASFT {
    wire [31:0] result;
    wire [31:0] source0;

    wire [4:0] shamt;
  stage 2 {
    source0 = GPR.read0(rs0);
    shamt = const__[4:0];

  };
  stage 3 {
    result = SFT0.sra(source0, shamt);
  };
  stage 4 {
  };
  stage 5 {
    null = GPR.write0(rd, result);
  };
};
micro_operation SLL on RG02ASFT {
    wire [31:0] source0;
    wire [31:0] source1;
    wire [31:0] result;
  stage 2 {
    source0 = GPR.read2(rs0);
    source1 = GPR.read3(rs1);
  };
  stage 3 {
    wire [4:0] shamt;

    shamt = source1[4:0];
    result = SFT0.sll(source0, shamt);
  };
  stage 4 {
  };
  stage 5 {
    null = GPR.write1(rd, result);
  };
};
-------- snip --------
micro_operation SRAI on RG02ASFT {
    wire [31:0] result;
    wire [31:0] source0;

    wire [4:0] shamt;
  stage 2 {
    source0 = GPR.read2(rs0);
    shamt = const__[4:0];

  };
  stage 3 {
    result = SFT0.sra(source0, shamt);
  };
  stage 4 {
  };
  stage 5 {
    null = GPR.write1(rd, result);
  };
};
micro_operation SLL on RG02BSFT {
    wire [31:0] source0;
    wire [31:0] source1;
    wire [31:0] result;
  stage 2 {
    source0 = GPR.read2(rs0);
    source1 = GPR.read3(rs1);
  };
  stage 3 {
    wire [4:0] shamt;

    shamt = source1[4:0];
    result = SFT0.sll(source0, shamt);
  };
  stage 4 {
  };
  stage 5 {
    null = GPR.write1(rd, result);
  };
};
-------- snip --------
micro_operation SRAI on RG02BSFT {
    wire [31:0] result;
    wire [31:0] source0;

    wire [4:0] shamt;
  stage 2 {
    source0 = GPR.read2(rs0);
    shamt = const__[4:0];

  };
  stage 3 {
    result = SFT0.sra(source0, shamt);
  };
```

```
    stage 4 {
    };
    stage 5 {
      null = GPR.write1(rd, result);
    };
  };
  micro_operation SLL on RG03BSFT {
      wire [31:0] source0;
      wire [31:0] source1;
      wire [31:0] result;
    stage 2 {
      source0 = GPR.read4(rs0);
      source1 = GPR.read5(rs1);
    };
    stage 3 {
      wire [4:0] shamt;

      shamt = source1[4:0];
      result = SFT0.sll(source0, shamt);
    };
    stage 4 {
    };
    stage 5 {
      null = GPR.write2(rd, result);
    };
  };
  -------- snip --------
  micro_operation SRAI on RG03BSFT {
      wire [31:0] result;
      wire [31:0] source0;

      wire [4:0] shamt;
    stage 2 {
      source0 = GPR.read4(rs0);
      shamt = const__[4:0];

    };
    stage 3 {
      result = SFT0.sra(source0, shamt);
    };
    stage 4 {
    };
    stage 5 {
      null = GPR.write2(rd, result);
    };
  };
    micro_operation SLL on RG03CSFT {
      wire [31:0] source0;
      wire [31:0] source1;
      wire [31:0] result;
    stage 2 {
      source0 = GPR.read4(rs0);
      source1 = GPR.read5(rs1);
    };
    stage 3 {
      wire [4:0] shamt;

      shamt = source1[4:0];
      result = SFT0.sll(source0, shamt);
    };
    stage 4 {
    };
    stage 5 {
      null = GPR.write2(rd, result);
    };
  };
  -------- snip --------
  micro_operation SRAI on RG03CSFT {
      wire [31:0] result;
      wire [31:0] source0;

      wire [4:0] shamt;
    stage 2 {
      source0 = GPR.read4(rs0);
      shamt = const__[4:0];

    };
    stage 3 {
      result = SFT0.sra(source0, shamt);
    };
    stage 4 {
    };
    stage 5 {
      null = GPR.write2(rd, result);
    };
  };
  micro_operation SLL on RG04CSFT {
      wire [31:0] source0;
      wire [31:0] source1;
      wire [31:0] result;
    stage 2 {
      source0 = GPR.read6(rs0);
      source1 = GPR.read7(rs1);
    };
    stage 3 {
      wire [4:0] shamt;

      shamt = source1[4:0];
      result = SFT0.sll(source0, shamt);
    };
    stage 4 {
    };
    stage 5 {
      null = GPR.write3(rd, result);
    };
  };
  -------- snip --------
  micro_operation SRAI on RG04CSFT {
      wire [31:0] result;
      wire [31:0] source0;

      wire [4:0] shamt;
    stage 2 {
      source0 = GPR.read6(rs0);
      shamt = const__[4:0];

    };
    stage 3 {
      result = SFT0.sra(source0, shamt);
    };
    stage 4 {
    };
    stage 5 {
      null = GPR.write3(rd, result);
    };
  };
  micro_operation BEQZ on RG01AJMP {
      wire [31:0] temp_pc;
      wire [31:0] offset;
      wire [31:0] source0;
    stage 2 {
      wire [31:0] ext_Const;
      wire [1:0] zero2;
      wire [29:0] temp_offset;

      source0 = GPR.read0(rs0);
```

```
      ext_Const = EXT00.sign(const__);
      zero2 = "00";
      temp_offset = ext_Const[29:0];
      offset = <temp_offset, zero2>;
      temp_pc = PC.read();
    };
    stage 3 {
      wire cond;
      wire [31:0] target;
      wire [3:0] flag;

      cond = source0 ==
      "00000000000000000000000000000000";
      <target,flag>
             = ALU0.add(temp_pc, offset);
      null = [cond] PC.write(target);
    };
    stage 4 {
    };
    stage 5 {
    };
  };
  micro_operation BNEZ on RG01AJMP {
      wire [31:0] temp_pc;
      wire [31:0] offset;
      wire [31:0] source0;
    stage 2 {
      wire [31:0] ext_Const;
      wire [1:0] zero2;
      wire [29:0] temp_offset;

      source0 = GPR.read0(rs0);
      ext_Const = EXT00.sign(const__);
      zero2 = "00";
      temp_offset = ext_Const[29:0];
      offset = <temp_offset, zero2>;
      temp_pc = PC.read();
    };
    stage 3 {
      wire cond;
      wire [31:0] target;
      wire [3:0] flag;

      cond = source0 !=
      "00000000000000000000000000000000";
      <target,flag>
             = ALU0.add(temp_pc, offset);
      null = [cond] PC.write(target);
    };
    stage 4 {
    };
    stage 5 {
    };
  };
  micro_operation J on RG01AJMP {
      wire [31:0] temp_pc;
      wire [31:0] offset;
    stage 2 {
      wire [1:0] zero2;
      wire [27:0] ext_const__;

      temp_pc = PC.read();
      zero2 = "00";
      ext_const__ = <const__, zero2>;
      offset = EXT1.sign(ext_const__);
    };
    stage 3 {
      wire [31:0] target;
      wire [3:0] flag;

      <target, flag>
             = ALU0.add(temp_pc, offset);
      null = PC.write(target);
    };
    stage 4 {
    };
    stage 5 {
    };
  };
  micro_operation JAL on RG01AJMP {
      wire [31:0] link;
      wire [31:0] temp_pc;
      wire [31:0] offset;
    stage 2 {
      wire [1:0] zero2;
      wire [27:0] ext_const__;

      temp_pc = PC.read();
      zero2 = "00";
      ext_const__ = <const__, zero2>;
      offset = EXT1.sign(ext_const__);


      link = PC.read();
    };
    stage 3 {
      wire [31:0] target;
      wire [3:0] flag;

      <target, flag>
             = ALU0.add(temp_pc, offset);
      null = PC.write(target);
    };
    stage 4 {
    };
    stage 5 {
      wire [4:0] reg_num;

      reg_num = "11100";
      null = GPR.write0(reg_num, link);
    };
  };
  -------- snip --------
  micro_operation ADD on RG01AALU {
      wire [31:0] source0;
      wire [31:0] source1;
      wire [31:0] result;
    stage 2 {
      source0 = GPR.read0(rs0);
      source1 = GPR.read1(rs1);
    };
    stage 3 {
      wire [3:0] flag;

      <result, flag>
            = ALU0.add(source0, source1);
    };
    stage 4 {
    };
    stage 5 {
      null = GPR.write0(rd, result);
    };
  };
  micro_operation ADDU on RG01AALU {
      wire [31:0] source0;
      wire [31:0] source1;
```

```
      wire [31:0] result;
    stage 2 {
      source0 = GPR.read0(rs0);
      source1 = GPR.read1(rs1);
    };
    stage 3 {
      wire [3:0] flag;

      <result, flag>
           = ALU0.addu(source0, source1);
    };
    stage 4 {
    };
    stage 5 {
      null = GPR.write0(rd, result);
    };
  };
  micro_operation ADDI on RG01AALU {
      wire [31:0] result;
      wire [31:0] source0;
      wire [31:0] source1;
    stage 2 {
      source0 = GPR.read0(rs0);
      source1 = EXT00.sign(const__);
    };
    stage 3 {
      wire [3:0] flag;

      <result, flag>
            = ALU0.add(source0, source1);
    };
    stage 4 {
    };
    stage 5 {
      null = GPR.write0(rd, result);
    };
  };
  micro_operation ADDUI on RG01AALU {
      wire [31:0] result;
      wire [31:0] source0;
      wire [31:0] source1;
    stage 2 {
      source0 = GPR.read0(rs0);
      source1 = EXT00.sign(const__);
    };
    stage 3 {
      wire [3:0] flag;

      <result, flag>
           = ALU0.addu(source0, source1);
    };
    stage 4 {
    };
    stage 5 {
      null = GPR.write0(rd, result);
    };
  };
  -------- snip --------
  micro_operation ADD on RG02AALU {
      wire [31:0] source0;
      wire [31:0] source1;
      wire [31:0] result;
    stage 2 {
      source0 = GPR.read2(rs0);
      source1 = GPR.read3(rs1);
    };
    stage 3 {
      wire [3:0] flag;

      <result, flag>
           = ALU0.add(source0, source1);
    };
    stage 4 {
    };
    stage 5 {
      null = GPR.write1(rd, result);
    };
  };
  -------- snip --------
  micro_operation MULT on RG01AMUL {
      wire [31:0] source0;
      wire [31:0] source1;
      wire [31:0] result;
    stage 2 {
      source0 = GPR.read0(rs0);
      source1 = GPR.read1(rs1);
    };
    stage 3 {
      wire [63:0] tmp_result;

      tmp_result
           = MUL0.mul(source0, source1);
      result = tmp_result[31:0];
    };
    stage 4 {
    };
    stage 5 {
      null = GPR.write0(rd, result);
    };
  };
  micro_operation MULTU on RG01AMUL {
      wire [31:0] source0;
      wire [31:0] source1;
      wire [31:0] result;
    stage 2 {
      source0 = GPR.read0(rs0);
      source1 = GPR.read1(rs1);
    };
    stage 3 {
      wire [63:0] tmp_result;

      tmp_result
          = MUL0.mulu(source0, source1);
      result = tmp_result[31:0];
    };
    stage 4 {
    };
    stage 5 {
      null = GPR.write0(rd, result);
    };
  };
  micro_operation MULT on RG02AMUL {
      wire [31:0] source0;
      wire [31:0] source1;
      wire [31:0] result;
    stage 2 {
      source0 = GPR.read2(rs0);
      source1 = GPR.read3(rs1);
    };
    stage 3 {
      wire [63:0] tmp_result;

      tmp_result
          = MUL0.mul(source0, source1);
      result = tmp_result[31:0];
    };
```

```
    stage 4 {
    };
    stage 5 {
      null = GPR.write1(rd, result);
    };
  };
  -------- snip --------
  micro_operation DIV on RG01ADIV {
      wire [31:0] source0;
      wire [31:0] source1;
      wire [31:0] result;
      wire [31:0] mod_result;
    stage 2 {
      source0 = GPR.read0(rs0);
      source1 = GPR.read1(rs1);
    };
    stage 3 {
      wire div_flag;

      <result, mod_result, div_flag>
           = DIV0.div(source0, source1);
    };
    stage 4 {
    };
    stage 5 {
      null = GPR.write0(rd, result);
    };
  };
  micro_operation DIVU on RG01ADIV {
      wire [31:0] source0;
      wire [31:0] source1;
      wire [31:0] result;
      wire [31:0] mod_result;
    stage 2 {
      source0 = GPR.read0(rs0);
      source1 = GPR.read1(rs1);
    };
    stage 3 {
      wire div_flag;

      <result, mod_result, div_flag>
          = DIV0.divu(source0, source1);
    };
    stage 4 {
    };
    stage 5 {
      null = GPR.write0(rd, result);
    };
  };
  micro_operation MOD on RG01ADIV {
      wire [31:0] source0;
      wire [31:0] source1;
      wire [31:0] result;
      wire [31:0] div_result;
    stage 2 {
      source0 = GPR.read0(rs0);
      source1 = GPR.read1(rs1);
    };
    stage 3 {
      wire div_flag;

      <div_result, result, div_flag>
           = DIV0.div(source0, source1);
    };
    stage 4 {
    };
    stage 5 {
      null = GPR.write0(rd, result);
    };
  };
  -------- snip --------
  micro_operation LB on RG01AMEM {
      wire [31:0] source0;
      wire [31:0] source1;
      wire [31:0] addr;
      wire [31:0] result;
    stage 2 {
      source0 = GPR.read0(rs0);
      source1 = EXT00.sign(const__);
    };
    stage 3 {
      wire [3:0] flag;
      <addr,flag>
             = ALU0.add(source0,source1);
    };
    stage 4 {
      wire addr_err;
      <result, addr_err> = DMAU.lb(addr);

    };
    stage 5 {
      null = GPR.write0(rd, result);
    };
  };
  micro_operation LH on RG01AMEM {
      wire [31:0] source0;
      wire [31:0] source1;
      wire [31:0] addr;
      wire [31:0] result;
    stage 2 {
      source0 = GPR.read0(rs0);
      source1 = EXT00.sign(const__);
    };
    stage 3 {
      wire [3:0] flag;
      <addr,flag>
             = ALU0.add(source0,source1);
    };
    stage 4 {
      wire addr_err;
      <result, addr_err> = DMAU.lh(addr);

    };
    stage 5 {
      null = GPR.write0(rd, result);
    };
  };
  micro_operation LW on RG01AMEM {
      wire [31:0] source0;
      wire [31:0] source1;
      wire [31:0] addr;
      wire [31:0] result;
    stage 2 {
      source0 = GPR.read0(rs0);
      source1 = EXT00.sign(const__);
    };
    stage 3 {
      wire [3:0] flag;
      <addr,flag>
             = ALU0.add(source0,source1);
    };
    stage 4 {
      wire addr_err;
      <result, addr_err>
                     = DMAU.load(addr);

```

```
1524     };
1525     stage 5 {
1526       null = GPR.write0(rd, result);
1527     };
1528   };
1529   micro_operation LBU on RG01AMEM {
1530       wire [31:0] source0;
1531       wire [31:0] source1;
1532       wire [31:0] addr;
1533       wire [31:0] result;
1534     stage 2 {
1535       source0 = GPR.read0(rs0);
1536       source1 = EXT00.sign(const__);
1537     };
1538     stage 3 {
1539       wire [3:0] flag;
1540       <addr,flag>
1541             = ALU0.add(source0,source1);
1542     };
1543     stage 4 {
1544       wire addr_err;
1545       <result, addr_err>
1546                         = DMAU.lbu(addr);
1547
1548     };
1549     stage 5 {
1550       null = GPR.write0(rd, result);
1551     };
1552   };
1553   micro_operation LHU on RG01AMEM {
1554       wire [31:0] source0;
1555       wire [31:0] source1;
1556       wire [31:0] addr;
1557       wire [31:0] result;
1558     stage 2 {
1559       source0 = GPR.read0(rs0);
1560       source1 = EXT00.sign(const__);
1561     };
1562     stage 3 {
1563       wire [3:0] flag;
1564       <addr,flag>
1565             = ALU0.add(source0,source1);
1566     };
1567     stage 4 {
1568       wire addr_err;
1569       <result, addr_err>
1570                         = DMAU.lhu(addr);
1571
1572     };
1573     stage 5 {
1574       null = GPR.write0(rd, result);
1575     };
1576   };
1577   micro_operation SB on RG01AMEM {
1578       wire [31:0] data;
1579       wire [31:0] base;
1580       wire [31:0] offset;
1581       wire [31:0] addr;
1582     stage 2 {
1583       data = GPR.read0(rd);
1584       base = GPR.read1(rs0);
1585       offset = EXT00.sign(const__);
1586     };
1587     stage 3 {
1588       wire [3:0] flag;
1589
1590       <addr, flag>
1591                 = ALU0.add(base,offset);
1592     };
1593     stage 4 {
1594       wire addr_err;
1595       addr_err = DMAU.sb(addr,data);
1596     };
1597     stage 5 {
1598     };
1599   };
1600   micro_operation SH on RG01AMEM {
1601       wire [31:0] data;
1602       wire [31:0] base;
1603       wire [31:0] offset;
1604       wire [31:0] addr;
1605     stage 2 {
1606       data = GPR.read0(rd);
1607       base = GPR.read1(rs0);
1608       offset = EXT00.sign(const__);
1609     };
1610     stage 3 {
1611       wire [3:0] flag;
1612
1613       <addr, flag>
1614             = ALU0.add(base,offset);
1615     };
1616     stage 4 {
1617       wire addr_err;
1618       addr_err = DMAU.sh(addr,data);
1619     };
1620     stage 5 {
1621     };
1622   };
1623   micro_operation SW on RG01AMEM {
1624       wire [31:0] data;
1625       wire [31:0] base;
1626       wire [31:0] offset;
1627       wire [31:0] addr;
1628     stage 2 {
1629       data = GPR.read0(rd);
1630       base = GPR.read1(rs0);
1631       offset = EXT00.sign(const__);
1632     };
1633     stage 3 {
1634       wire [3:0] flag;
1635
1636       <addr, flag>
1637             = ALU0.add(base,offset);
1638     };
1639     stage 4 {
1640       wire addr_err;
1641       addr_err = DMAU.store(addr,data);
1642     };
1643     stage 5 {
1644     };
1645   };
1646   -------- snip --------
1647 }
```