Static resource models for code generation of embedded processors

Static Resource Models for Code Generation of Embedded Processors

Qin Zhao

Static Resource Models for Code Generation of Embedded Processors

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
Rector Magnificus, prof.dr. R.A. van Santen, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op donderdag 23 oktober 2003 om 16.00 uur

door

Qin Zhao

geboren te Pinglang, Gansu, China

Dit proefschrift is goedgekeurd door de promotoren:

prof.Dr.-Ing. J.A.G. Jess
en
prof.dr.ir. J.L. van Meerbergen

Copromotor:
dr.ir. B. Mesman

Druk: Universiteitsdrukkerij, Technische Universiteit Eindhoven

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Zhao, Qin

Static resource models for code generation of embedded processors / by
Qin Zhao. -Eindhoven : Technishe Universiteit Eindhoven,
2003.
Proefschrift. - ISBN 90-386-1775-5
NUR 958
Trefw.: ingebedde systemen / compilers / scheduling / digitale systemen ; CAD /
digitale signaalverwerking ; software.
Subject headings: embedded systems / program compilers / processor scheduling
/ instruction sets / storage allocation.

Acknowledgments

First of all I would like to thank Prof. Jochen Jess, who offered me the opportunity to start my Ph.D. study in the Information and Communication Systems(ICS) Group, and supported me during my stay in this group.

I am grateful to Dr. Koen van Eijk, who guided me into this field, showed me the versatile research directions and supported me with the wonderful framework FACTS. I am also grateful to Dr. Twan Basten for his help in mathematics, his careful corrections in our publications and our corporation in Ozone project. Many thanks attribute to Dr. Bart Mesman for his coaching, his inspiring ideas and valuable discussions. His help resulted in the improvement in my presentations and a number of cooperative publications, and finally this thesis.

Furthermore, I want to express my gratitude to Prof. Ralf Otten for his support, Prof. Jef van Meerbergen, Prof. Henk Corporaal, Prof. Patrick Groeneveld and Prof. Patrick Dewilde for their guidance and advices to improve this thesis.

Many thanks owe to my previous and present colleges in the ICS Group. In particular, to Dr. Geert Janssen for his guidance in programming, to Dr. Jeroen Rutten, Dr. Etienne Jacobs and Dr. Michel Berkelaar for their help in the adaptation of dutch culture, to my previous roommate Dr. Carlos Alba Pinto for his corporation and friendship, and to Lia de Jong, Marja de Mol-Regels, Rian van Gaalen and Oege Koopmans for their kindness. I am also grateful to Dr. Marc Geilen for our corporation in Ozone project, and Dirk-Jan Jongeneel for putting my computer to work properly. Finally, I want to thank all the people in the ICS group, Jurjen Westra, Srinath Naidu, Aleksander Beric, Peter Poplavko, Meng Zhao, Kasia Leijten-Nowak, Amir Ghamarian, Sander Stuijk, Hamed Fatemi etc.

Last but not least, I would like to thank my parents a lot. Even they are far away, they are always close to me in all aspects of life all the time. I want to give my special thanks to my husband Dr. Bin Yin for his love, care, encouraging, interesting discussions during my whole Ph.D. research and study.

Summary

Embedded systems play a more and more important role in our daily lives. Their market size is about 100 times the size of the desktop market . Often these systems must meet strict constraints regarding performance, area, cost, power consumption and real-time behavior. In order to meet the design constraints of embedded systems, it is convenient to integrate the entire system on a single chip, which is the so-called System-on-Chip approach. The processor cores embedded may be off-the-shelf general-purpose processor cores, Application Specific Instruction-Set Processors (ASIPs) or Application Specific Integrated circuits (ASICs). An ASIP is a processor that is designed to efficiently execute functions from a specific application domain.

Embedded processors have large variations in their architectures regarding performance, area and power requirements. Especially ASIPs employ a lot of irregularities in their architectures, e.g. heterogeneous register sets, a small number of specialized registers, specialized functional units, restricted connectivity, limited addressing and highly irregular data paths. The instruction set for an ASIP is usually highly encoded, which is designed with minimum number of encoding bits to constrain power consumption. The use of conventional code generation techniques and compilers often produces very inefficient code for these architectures. Therefore, in order to meet the given constraints, the critical parts of programs are written in assembly code by hand. This heavily reduces the portability and maintainability of the generated code. Due to the increasing complexity in digital signal processing, high-level compilation is desirable.

Code generation, the back-end of compilation, as performed in sequential phases for general-purpose processors, is not adequate for embedded processors. This is because decisions in one phase will affect the search space of the rest of the phases. Consider a highly-encoded instruction set that imposes severe limits on the amount of parallelism; on the one hand, if instruction selection is performed prior to scheduling, the optimal schedule can easily be eliminated as a result of the choices made during instruction selection; on the other hand, if scheduling is performed first, the available instructions may not be able to implement the schedule. Furthermore, distributed register files with limited connectivity make it more

difficult to meet the register file constraints. All these characteristics require a unified code generation for efficient compilation.

In this thesis, the methodology of the Static Resource Model (SRM) is proposed for modeling the highly-encoded instruction sets of embedded processors. These constraints are combined with the resource and timing constraints in such a way that a unified environment for constraint description is specified. The SRM approach is applied to a reconfigurable instruction set processor. The reconfigurable part of this processor lies in the instruction decoder, so that the instruction set constraints can be modeled efficiently through constructing the SRMs for the different configurations. The SRM approach also presents an efficient solution for the instruction set design since all the constraints originating from instruction sets are represented as static resources, which can be used for quick evaluation of the performance. A modification of these constraints has a direct effect on the performance and code size.

The SRM approach is extended to cope with the limited address range (LAR) register file architecture. In this architecture, although a functional unit is physically fully connected to a register file, it reads from and writes to only a subset of registers in the register file, consequently reducing the number of encoding bits. Functional units may access different subsets of a register file. These subsets may overlap to allow the communications between those functional units. The additional range assignment phase can be alleviated by integrating the limited address constraints with the value lifetime conflicts, resulting in a uniform conflict graph for register allocation.

Samenvatting

Ingebedde systemen spelen een steeds belangrijkere rol in ons dagelijkes leven. Hun afzetmarkt is pakweg honderd keer zo groot als die van desktops. Deze systemen dienen veelal strikte beperkingen na te leven met betrekking tot rekenkracht, oppervlakte, kosten, energie verbruik, en tijdsgedrag. Om deze ontwerpbeperkingen na te leven is het handig om het gehele systeem te integreren op een enkele chip, de zogenaamde “system op chip” benadering. De ingebedde processors kunnen algemeen toepasbare processors zijn, applicatiespecifieke instructieset processors (ASIPs), of applicatiespecifieke geïntegreerde circuits (ASICs). Een ASIP is een processor ontworpen om efficiënt functies uit te voeren uit een specifiek applicatie domein.

Ingebedde processors vertonen een grote variatie aan architecturen met betrekking tot vereisten omtrent rekenkracht, oppervlak en energieverbruik. Met name ASIPs passen in hun architectuur een grote hoeveelheid irreguliere zaken toe, zoals heterogene register sets, een klein aantal gespecialiseerde registers, gespecialiseerde functionele eenheden, beperkte verbindingen, beperkte adressering, en zeer irreguliere data paden. De instructieset van een ASIP is gewoonlijk in een sterk gecodeerd formaat, ontworpen met een minimaal aantal code bits teneinde het energieverbruik te berperken. Conventionele codegeneratie technieken en verlaters produceren veelal zeer inefficiënte code voor deze architecturen. Om aan de ontwerpbeperkingen to voldoen worden kritische delen van programma’s met de hand geschreven in machine code. Dit maakt het erg lastig om de gegenreerde code over te dragen en te onderhouden. Door de toenemende complexiteit van in digitale singaalbewerking is hoogniveau verlating zeer gewenst.

Code generatie, het laatste deel van verlating zoals toegepast in sequentiële fasen voor algemeen toepasbare processors, in niet adequaat voor ingebedde processors. Dit komt doordat beslissingen genomen in een vroege fase, de zoekruimte beperken voor latere fasen. Neem een sterk gecodeerde instructieset in ogenschouw die ernstige beperkingen oplegt aan de hoeveelheid parallellisme. Indien aan de ene kant instructieselectie wordt uitgevoerd voorafgaand aan tijdstoekenning, kan de optimale tijdstoekenning verdwenen zijn als gevolg van keuzes gemaakt tijdens instructieselectie. Als aan de andere kant tijdstoekenning als eerst

wordt uitgevoerd, kunnen de beschikbare instructies mogelijk niet de tijdstoekenning implementeren. Bovendien maken gedistribueerde register files met beperkte verbindingen het moeilijk te voldoen aan register file capaciteit. Al deze karakteristieken maken een unificatie voor code generatie nodig voor een efficiënte verlating.

In dit proefschrift wordt een methodologie voorgesteld waarbij het Statisch Resource Model (SRM) wordt gebruikt om sterk gecodeerde instructiesets te modelleren. Deze beperkingen worden gecombineerd met de resource– en tijdsbeperkingen teneinde een omgeving te creëren met een unificatie voor het beschrijven van beperkingen. De SRM benadering wordt toegepast op een herconfigureerbare instructieset processor. Het herconfigureerbare deel van de processor is bevat in de instructiedecoder, en wel op een dusdanige manier dat de instructieset beperkingen efficiët gemodelleerd kunnen worden door het opstellen van de SRMs voor de verscheidene configuraties. De SRM benadering vormt ook een goede oplossing voor het *ontwerpen* van instructiesets omdat alle beperkingen voortvloeiend uit een instrucieset worden gerepresenteerd als statische resources, die worden aangewend om tot een snelle evaluatie te komen van de rekenkracht en de code grootte.

De SRM benadering wordt uitgebreid teneinde om te kunnen gaan met de beperkt adressering van register files (LAR). Hoewel in zo'n architectuur een functionele eenheid verbonden is met de volledige register file, leest en schrift deze eenheid effectief slechts een deel van de register file, teneinde de benodigde code bits te reduceren. Verschillende functionele eenheden bereiken verschillende delen van de register file. De verschillende delen kunnen echter overlappen om communicatie te berwerkstelligen tussen de functionele eenheden. De vertaal fase voor het toekennen van delen van register files kan worden verlicht door de adres beperkingen samen met de leeftijdsconflicten te integreren in een uniform conflict graaf model voor het toekennen van registers.

Contents

Chapter 1

Introduction

1.1 Design of embedded processors

Embedded systems [25] are highly specialized, often reactive, sub-systems that provide, unnoticed by the user, information processing and control tasks to their embedding system. An embedded system is a computer system with hardware (application specific integrated circuit) and software (programmable processor and application code) specifically designed for a particular application with real-time constraints. Examples of embedded systems include automobile engine-control units, laser printers, electromechanical units in fax/data modems, cellular telephones and micro ovens, etc. These systems often must meet strict constraints regarding performance, size, cost and power consumption and real-time behavior. Low cost is necessary for high volume production and low power consumption is necessary for portable electronic equipments that are battery-operated. One of the main characteristics of embedded systems is the real-time constraint, which means that the application has to take place within a certain time that is enforced by the environment. The software component of embedded systems is referred to as the *embedded software*, and the processor on which the software is executed is referred to as the *embedded processor*.

In order to meet the design constraints of embedded systems, it is convenient to integrate the entire system on a single integrate circuit (IC), the so-called System-on-Chip (SoC) approach. This is enabled by the current deep sub-micron processing technology. Figure 1.1 illustrates a possible system-on-chip architecture, which consists of a processor core, program ROM/RAM, memory, application specific circuitry and peripherals. The processor core may be an off-the-shelf general-purpose processor core, an Application Specific Instruction-Set Processor (ASIP) or an Application Specific Integrated Circuits (ASIC). An ASIP is a processor that is designed to efficiently execute functions from a specific appli-

cation domain, while an ASIC is a non-programmable or a partly-programmable integrated circuit for a single task or an application.

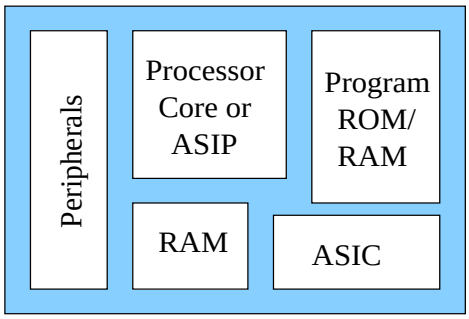


Figure 1.1: A heterogeneous system-on-chip architecture

Given the System-on-Chip approach, a systematical way of implementing an application on this system may follow the *hardware-software co-design* methodology, depicted in Figure 1.2 with several steps:

- determine those parts of the behavioral description that are to be implemented in hardware, and those parts that are to be implemented in software,
- generate the custom circuitry and embedded software components using hardware synthesis and compilation techniques, respectively,
- simulate the system in order to determine whether all constraints (e.g. correctness and timing constraints) have been satisfied. If this is the case, then the design process ends; otherwise, determine a different hardware/software partitioning of the behavioral description, and reiterate te design process.

Although incorporating a complete system on a single IC may improve performance, cost and power consumption requirements, such a high level of integration constrains the size of the system components. As a result, both hardware and software components should be designed with minimum size in mind. This implies that not only the ASIC and embedded processor need to be designed within the size constraints of the given IC, but also the remainder of the system components. This includes the program ROM/RAM, which stores the application code that is to be executed on the embedded processor. As a result, the application code size has to be kept small. For example, for GSM Enhanced Full Rate (EFR) speech codec being implemented in StarCore processor [67], the compiler optimized for code size will generate 35884 bytes code. For DSP56600 [22], the code written in

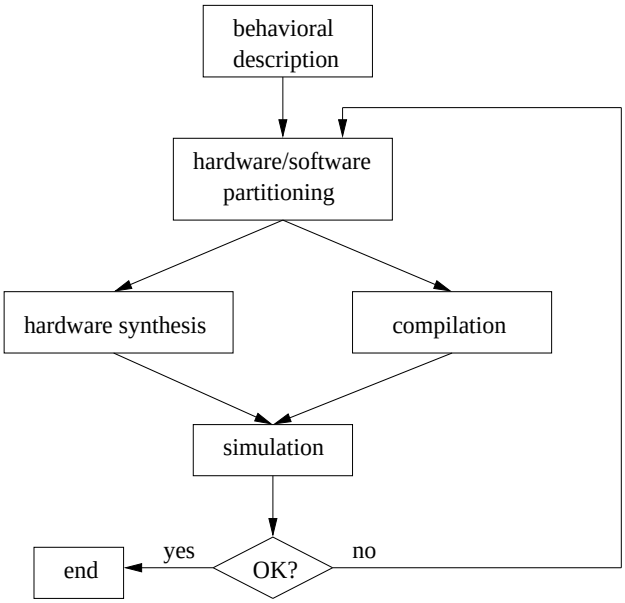


Figure 1.2: Hardware-software co-design methodology

assembly is about 25788 bytes. Reducing the code size of the application implies the necessity of developing an efficient instruction set architecture (ISA) and efficient compiler which can be tuned towards the application to optimize the code for minimum size.

In this thesis, we aim at application specific processor cores and the corresponding efficient compilation for small code size purpose on embedded systems.

1.1.1 Flexibility and efficiency of embedded programmable processors

Programmable processors can be roughly distinguished by two categories: stand-alone and embedded cores. General-purpose processors are usually stand-alone, which are responsible for the whole computation and control, although some general-purpose CPUs targeting small code size can also be used as embedded cores. On the contrary, DSPs and ASIPs are commonly used as embedded core components for digital signal processing and specific applications, while there are also general-purpose DSPs which are capable of large amount of computations, such as multimedia processors used for image and video processing. In this thesis, we focus on the embedded programmable processors for specific ap-

plications. The choice of a suitable processor to be integrated into an embedded system mainly depends on the application domain. They can be classified into three classes according to the flexibility and efficiency: general-purpose CPU cores, DSP cores and ASIPs. Roughly speaking, the more flexible a processor is, the less efficient it is for the application. This is illustrated in Figure 1.3. In this figure, a white ellipse symbolizes an embedded system, such as the one depicted in Figure 1.1, with gray ellipses representing all kinds of processor cores that can be embedded within this system. Squares correspond to the tasks or applications to be implemented on these cores, with the size indicating the size of the target application domain.

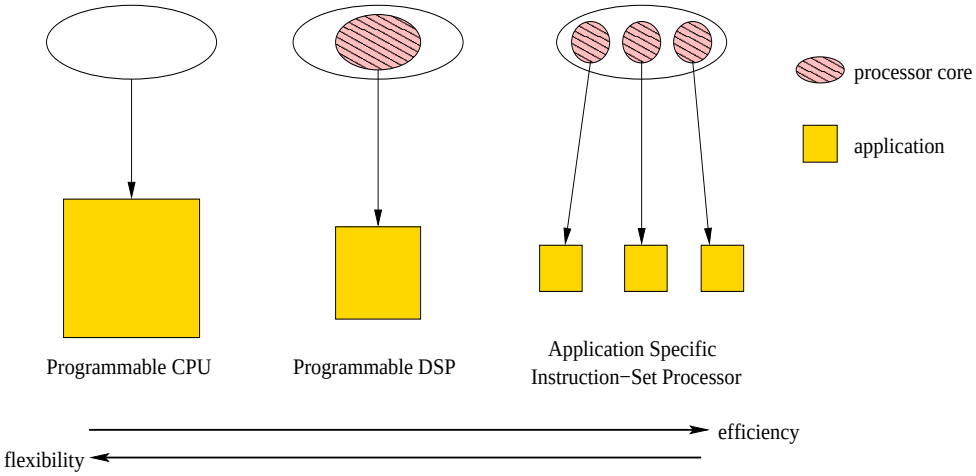


Figure 1.3: Flexibility and efficiency of embedded processors

- General-purpose CPU cores. The most flexible embedded processors are general-purpose programmable CPU cores, such as MIPS R3000, MIPS R4000 and ARM. On these cores every task or application can basically be implemented. They usually have a basic set of instructions with the same size and the execution of instructions can be pipelined in order to achieve a high throughput. In addition, they usually have caches that may cause cache misses. An operating system may be present that takes run time decisions. A typical example is the ARM RISC core. The ARM7 core architecture is specially designed for low power consumption. It is a 32-bit processor, although it's instruction format can be switched between a length of 16 or 32-bit.

- Digital signal processors (DSPs). They have been designed for arithmetic-intensive signal processing applications. Their instruction sets are tuned for fast execution of algorithms such as digital filtering and Fast Fourier Transform. This is supported by special hardware, e.g. MAC (Multiply-ACcumulator), address generation units and bit manipulation units. In order to allow for fast signal processing, they contain a degree of instruction level parallelism (ILP). In addition, they have special-purpose registers for shorter delay and fewer encoding bits. All these characteristics make it difficult for efficient compilation.

- Application specific instruction-set processors (ASIPs). This class of processors are domain-specific and serve only a very narrow range of applications. In order to be tuned for different applications, these processors are parameterizable. The basic architecture template is fixed, but it can be customized by setting a number of different parameters, such as instruction encoding, register files size, register file connectivity with functional units, availability of special hardware components. Since these parameters are orthogonal to each other, a large number of different configurations may be available. They are also synthesizable in the sense that detailed architecture specifications can be obtained by synthesizing the performance requirements and resource requirements. Consequently, retargetable compilers are needed for these different configurations to obtain portable code with efficiency.

Selecting which type of processors to use depends heavily on the application domain. In addition, in each type of processor there is also a large variation from one architecture to another. Obviously, to obtain high performance, small code size and low power design with reusable, portable, maintainable code, an efficient compilation is helpful in speeding up the design period.

The instruction set is the interface between a programmer and the processor. To obtain small code size, instruction set is usually highly encoded for embedded processors. In the next sections, we mainly discuss the different types of instruction sets regarding to code type, encoding style and compression technique.

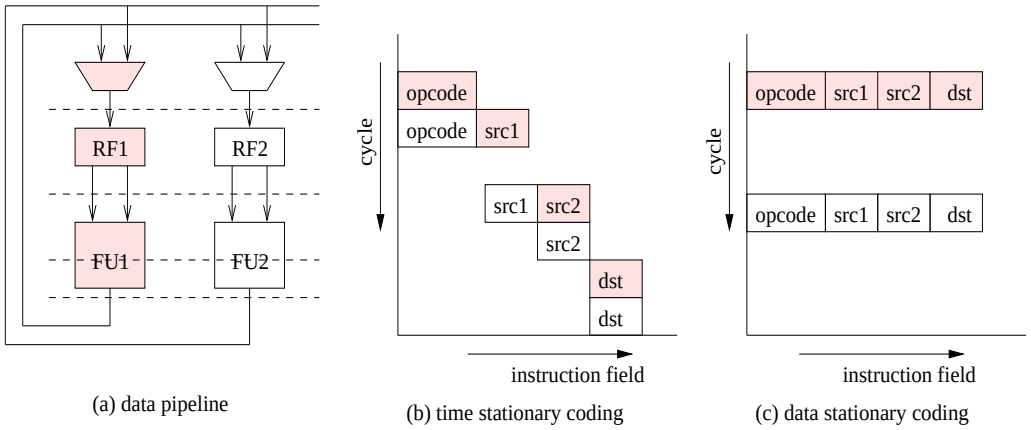
1.1.2 Instruction encoding in embedded processors

Embedded processors have large variations in their architecture according to the performance, area and power requirements. Correspondingly, their instruction sets also differ to a large extent. In this section, instruction encoding is reviewed from two aspects: the code type of an instruction set and the instruction format. *Pipelining* is an implementation technique whereby multiple instructions

are overlapped in execution; it takes advantage of parallelism that exists among the actions needed to execute an instruction. Based on the control of operations in the data pipeline, a distinction can be made between *time-stationary* coding and *data-stationary* coding, depicted in Figure 1.4.

- In the case of *time-stationary* coding, every instruction that is part of the processor's instruction set controls a complete set of data pipeline stages that each has to be executed in a single machine cycle, i.e. the data pipeline of the processor is visible in the machine code. A single operation traversing the data pipeline may process several of these stages. Either a programmer or a compiler has to maintain the data pipeline and make it explicit in the program.

- In the case of *data-stationary* coding, every instruction that is part of the processor's instruction set controls a complete sequence of operations that have to be executed on a specific data item, i.e. the opcode, register source and destination addresses are in the same instruction word. Once the instruction has been fetched from the program memory and decoded, the processor controller hardware will make sure that each part is executed in the correct machine cycle.



(a) data pipeline (b) time stationary coding (c) data stationary coding

Figure 1.4: Time stationary vs. data stationary encoding

The instruction width of time-stationary encoding is relatively large, since every stage of a pipeline has to be explicitly expressed. In addition, it requires hardware control to keep the consistency of the functional units and the registers

addressed in one instruction. The advantage is that when an interrupt happens, the results in the previous stages do not need to be saved. The data-stationary encoding is more concise and easy to program, and there is no need for centralized control resource. However, it is costly for interrupts in the sense that every intermediate result in the previous instruction has to be saved. This is because when an interrupt happens for one instruction, the previous one might not be finished yet and still be in the different stages of the pipeline.

Based on the instruction encoding format, a distinction can be made between an orthogonal instruction format and an encoded instruction format.

- An *orthogonal format* consists of fixed control fields that can be set independently from each other. Each control field usually consists of opcode and operand fields for controlling the execution of one operation on certain functional unit. For example, very long instruction word (VLIW) processors have an orthogonal instruction format, such as the one in Figure 1.5. The instruction bits within every control field may additionally have been encoded to reduce the field's width.

- In the *encoded format*, the instruction bits used to control one operation depend on the combinations of operations encoded in parallel with it. Not only the opcode field, but also the operand field can be different for different combinations. The correct interpretation can be deduced from the value of specially designated bits in the instruction word. One example is the ADSP-21xx instruction set depicted in Table 1.1.

1.1.3 Code compression techniques

Encoding may limit the amount of *instruction-level parallelism* (ILP) offered by the processor. The less restrictive an instruction set is, the more ILP can be obtained. In image processing and multimedia applications, often a large amount of ILP is required, so VLIW architectures such as TMSC3206x [71] or Trimedia [59] are a natural choice. In order to reduce the instruction width, some compression is performed for the instruction encoding.

Traditional VLIW architectures use rigid instruction field and / or special pack and unpack operations [62]. For example, in Cydra5 instruction format depicted in Figure 1.5, the MultiOp instruction format is defined with 256 bits for encoding totally six operations on six functional units plus an additional one to control the Instruction Unit and other miscellaneous operations. Each of the seven slots looks like a conventional RISC instruction. They are not equal in width. Typically one slot consists of an opcode, two source register specifiers, one destination register specifier and one predicate specifier. In addition, the UniOp instruction format,

which allows only a single operation to be initiated, is defined and multiple UniOp instructions fit into a 256 bit container. Extra bits are useful in indicating which functional unit is being executed.

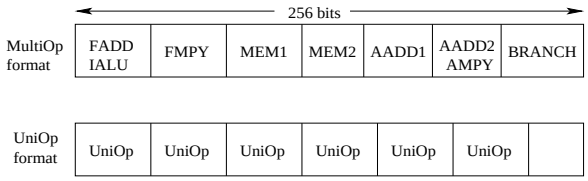


Figure 1.5: Cydra5 instruction format

In typical VLIW architectures, each control field corresponds to a particular functional unit. If that functional unit is idle on a particular clock cycle, a NOP would be placed in that functional unit's instruction slot. Unlike traditional VLIW architectures, recent commercial VLIW architectures allow more flexible instruction combinations to reduce the code size generated by NOP instructions while being able to exploit ILP effectively [26]. For example, both TMS320C6x DSP [71] and Lucent Star Core SC140 DSP [67] adopt a VLES (Various Length Execution Set) idea to achieve high code density while minimizing cost. There are two kinds of instruction packets in the code: *fetch packet* and *execute packet*, as illustrated in Figure 1.6 (a) and (b). In TMS320C6x, eight instructions are fetched at a time, which consists of one fetch packet. Fetch packets are aligned on 256-bit boundaries. The execution of individual instructions is partially controlled by one bit in each instruction (bit 0). This bit determines whether this instruction executes in parallel with the following instruction. If this bit is one, then this parallelism is allowed. All instructions executing in parallel constitute an execute packet. In the Lucent Star SC140 DSP, two MS (most significant) bits are used for the determination of an execute packet, or a prefix is used to indicate how many instructions constitute the execute packet. Several compression techniques [82], [48], [44] are developed for further reducing the instruction width. In [82], a scheme for block-based decompression in response to dynamic demands is presented. They consider a number of compression algorithms, and conclude that a Huffman code [40] is most effective. Liao et al. [48] develop a dictionary approach to reduce the code size for DSPs. In [44], a binary arithmetic code driven by a Markov model is proposed.

Speed and code size requirements for typical telecommunication and consumer applications make it necessary to design efficient ASIPs that have a relatively high degree of compression. It is a challenge to design an instruction set

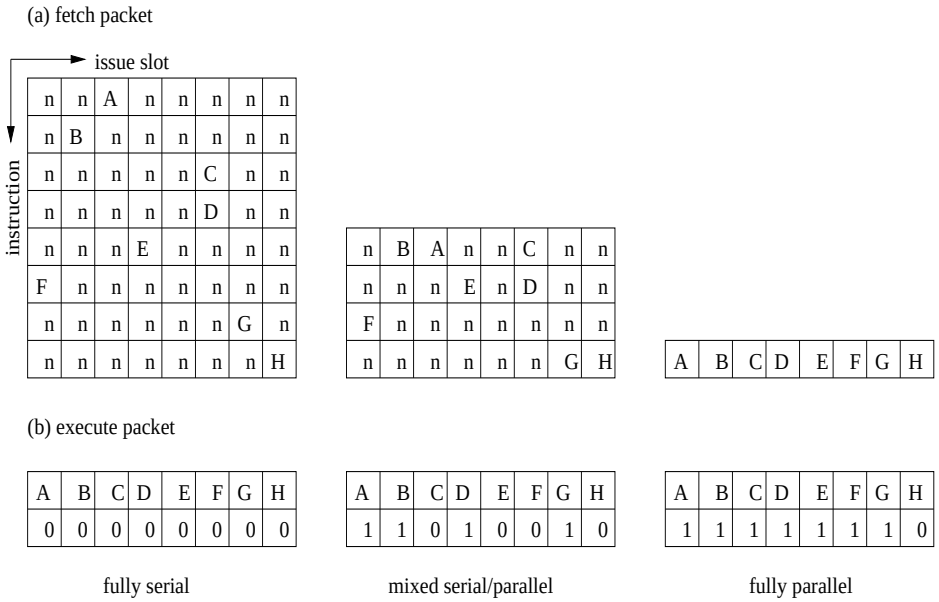
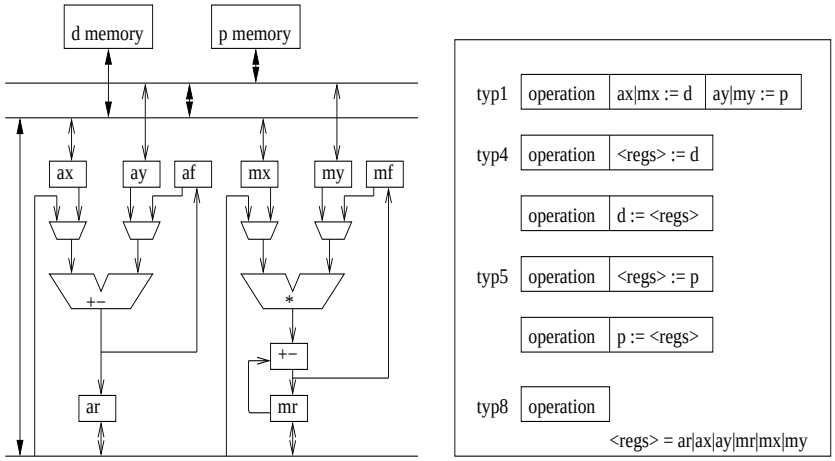


Figure 1.6: Fetch packet and execute packet instruction format

for an ASIP that can be encoded using a restricted number of instruction bits, while still offering a sufficient degree of parallelism for critical functions in the target applications. One typical example is to restrict the available number of registers for storing the value produced by one operation which is executed in parallel with other operations. Figure 1.7 (a) and (b) illustrate the data path and instruction formats of the ADSP-21xx family from Analog Devices respectively. There are four instruction formats, allowing for different combinations of operations being executed in parallel. The register usage for a functional unit also depends on the format.

The first instruction format is shown in detail in Table 1.1. Columns show different encoding fields in this format. As we can see, registers are roughly divided into groups X, Y, F and R. Two load operations from double memory banks D and P can be executed in parallel with an arithmetic operation, but they can only store the loaded values into X and Y registers separately. Although there is an ALU and a multiplier in the data path, there is only one arithmetic field. The arithmetic operation reads left source operand from Y or F groups, and reads right source operand from X or R groups. It writes the destination operand to R groups. Each group or register file contains a small number of registers. For example, MX register file contains registers $MX0$ and $MX1$.



(a) ADSP−21xx data path (b) ADSP−21xx instruction format

Figure 1.7: Data path and instruction format of ADSP-21xx

Table 1.1: Part of ADSP-21xx instructions

instruction	memory operation		arithmetic operation			
ALU/MAC	PM	DM	type	opcode	Y	X
with	00:AY0	00:AX0	0:MAC	$\cdots$	00:MY0	000:MX0
DM/PM	01:AY1	01:AX1		0100:$x * y_{ss}$	01:MY1	001:MX1
	10:MY0	10:MX0		0101:$x * y_{su}$	10:MF	010:AR
	11:MY1	11:MY1		$\cdots$		011:MR0
						$\cdots$
			1:ALU	$\cdots$	00:AY0	000:AX0
				0011:$x + y$	01:AY1	001:AX1
				1111:$abs(x)$	10:AF	010:AR
				$\cdots$		$\cdots$

Another way of keeping the instruction width small is exploited in the REAL DSP by using Application Specific Instruction (ASI) [83]. This architecture allows efficient encodings of many operations. In this DSP, small instruction words (e.g. 16 bits) are used for frequently occurring combinations of operations. Up to 256 VLIW instructions of 96 bits in a lookup table can be triggered by the 16 bits small instruction words, as illustrated in Figure 1.8. For memory fetching, indirect addressing can be applied to reduce the code size. When data are fetched from memory, the traditional way is to specify a memory address of e.g. 16 bits. A more efficient way is to store a base address of 16 bits inside the processor, and

specify an offset of e.g. four bits in the instruction word.

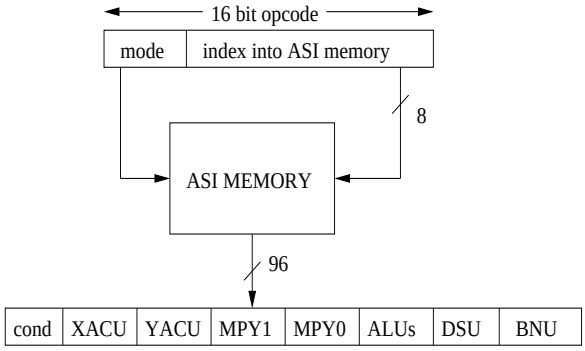


Figure 1.8: Application Specific Instruction (ASI) in REAL processor

Different instruction encodings, especially the highly-encoded instruction format, raise the difficulty for compilers to generate efficient code for different applications. Some adaptations have to be applied to the conventional software compilation process for application specific processors.

1.2 Compilers for embedded processors

The software compilation process illustrated in Figure 1.9 is usually used as the basis for the compilation for ASIPs. The starting point of a software compilation process is an application program in an *algorithmic specification language*, usually in C. It is translated into an *intermediate representation (IR)*, by means of a language-dependent front-end. Well known IRs for representing an algorithm include the *static single assignment form (SSA form)* [20], and the *control/data flow graph (CDFG)* [51]. The software compilation process is traditionally divided into high-level optimization and back-end compilation. In high-level optimization, a data-flow analysis is carried out to determine all required data dependencies in the algorithm and processor-independent optimizations are performed to reduce the number of operations, and increase the ILP of the description. The set of optimization includes common subexpression elimination, dead code removal, constant propagation and folding, etc, [3]. The back-end performs actual *code generation*, which maps machine-independent IR to machine-specific assembly instructions.

While compilers based on this flow generally generate satisfactory results for general-purpose processors, they are insufficient for ASIPs and DSPs [89]. For

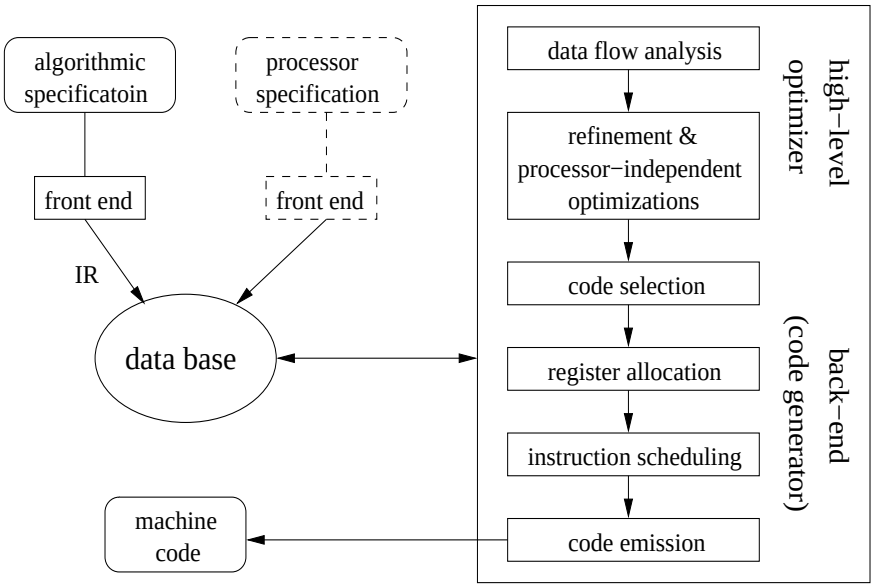


Figure 1.9: Overview of the compilation

parameterizable and synthesizable ASIPs, the compilers have to be *retargetable* to different processor configurations. The general treatment is that a processor specification is written in a machine description language (MDL) and the processor structure together with the instruction set architecture (ISA) information, which are extracted from the machine description file (MDF) written in MDL, are integrated in the data base for the back-end of the compiler.

1.2.1 Code generation in compilation

Code generation usually consists of three processes: code selection, register allocation and instruction scheduling. Each process corresponds to a NP-complete problem [31], thus heuristics are often applied for fast compilation. In order to further obtain run-time efficiency, they are usually performed in sequential phases in general-purpose processors.

- Code selection. The operations in the algorithmic model are bound to register transfers or partial instructions, supported by the target processor's instruction set. Multiple operations can be mapped into one partial instruction for optimal code size considerations. Usually tree pattern matching and dynamic programming are used to perform this process. Many DSPs have a

MAC unit for speeding up the processing, which can be encoded in one instruction and executed in one clock cycle. In order to exploit this function, the DFG has to be covered by the pattern of the MAC unit. An example is illustrated in Figure 1.10.

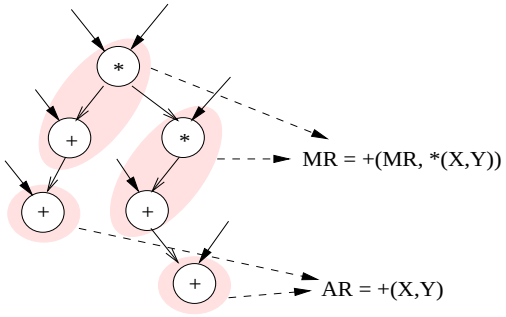


Figure 1.10: Code selection

- Scheduling. This process decides about the clock cycle assigned to an instruction. For DSPs, parallelism is the main concern in this process, in which data dependencies, latency, delays, the number of resources and pipeline effects are taken into account. Minimizing the execution time or obtaining maximum throughput in case of loop kernels is the optimizing objective.

- Register allocation. This process determines where the results of computations are stored, which can happen in registers or in memories. For low power considerations, we keep the memory accesses as low as possible and store values with non-overlapping lifetimes into one register. In case the number of available registers is exceeded, extra instructions are used to spill and reload values to and from memory.

With multiple functional units and more and more resources included in the architecture, and more complicated techniques, such as clustering being applied, additional problem, e.g. functional unit assignment, register file assignment, cluster assignment, bus assignment, etc, need to be solved. Performing code generation in sequential phases with heuristics generally produces satisfactory results for general-purpose processors, while it is frequently insufficient for embedded processors. This is mainly due to the following reasons:

- In most cases, fast heuristics are used for code generation. This is due to the demand for fast compilation for general-purpose processors. Since heuristics explore only a small part of the solution space of a code generation problem, the code quality may be compromised. For embedded processors, where the efficiency of the generated code is of major concern, possibly slower code generation techniques capable of exploring more solutions are a better choice.

- The decomposition of code generation into code selection, register allocation and instruction scheduling phases also affects the code quality, since these phases are mutually dependent. Ideally, all code generation phases therefore should be executed simultaneously in order to avoid an early decision which restricts the search space inappropriately. However, such a phase coupling approach is difficult to implement and time-consuming as well. In practice, most research concentrates on partially coupled code generation.

- Embedded processors often have irregular architectures, because they are highly optimized for certain objectives. Standard code generation techniques hardly take into account the special architecture features of embedded processors. Therefore, a powerful machine description language which can describe those features as well as a retargetable compiler which can deal with the constraints arising from those features are necessary for embedded processors.

For these reasons, integrating the phases as much as possible and building a uniform code generation in a compiler is a high demand for application specific processors. In addition, in order to generate reusable, maintainable and portable code, the compiler should also be retargetable for different architecture configurations.

1.2.2 Retargetability

The increasing use of software in embedded systems results in an increased flexibility from a system designer's point of view. However, different types of processor typically suffer from a lack of supporting tools. The major problem is that the target architecture is not fixed beforehand.

Compilation tools for synthesizable ASIPs must be easily adaptable to different processor architectures. This is essential to cope with the large degree of architecture variation. Moreover, market pressure results in increasingly shorter time-to-market of processor architecture. In this context, retargetable compilation

is the only solution to provide system designers with supporting tools. Retargetable compilers avoid the need to write different compilers for different configurations. In addition, they can help to determine the best configuration for given application, which is quite useful in making hardware and software trade-offs during system design.

Most of the target architectures are required to be optimized not only for performance, but also for area and cost. Since applications written in software are translated into processor instructions, most of the constraints arise due to encoding of instructions. Such constraints are difficult to capture in both and structural machine description languages (MDLs). In the first kind of MDL, the machine description is based on the ISA description, while in the second kind the machine description is extracted from the description of the processor's data path. Basically the constraints can be grouped into two kinds.

- Operation ILP constraints. These constraints describe the a set of operations that can or cannot be issued in parallel. For example, in VLIW architectures, the issue slots constraint defines that operations in a certain issue slot cannot be executed in parallel. In several DSPs, only one arithmetic operation such as *add* or *mul* can be executed in parallel with two memory operations, although in the data path there are one functional unit *ALU* and one functional unit *Multiplier* available in parallel.

- Operand ILP constraints. Those constraints describe how registers should be assigned to operands of operations issued in parallel. These constraints may or may not affect the data flow. For example in ADSP-21xx family, if two values are loaded from D memory and P memory simultaneously with an arithmetic operation, the loaded values can only be stored at AX (MX) or AY (MY) register files.

Besides the ILP constraints enforced by the MDL, the architecture itself contains limited resources, e.g. functional units, buses, register file ports, etc, which will also limit the amount of ILP. For all the above mentioned hardware limitations, a constraint-based approach seems a proper way to perform the retargetable compilation. Using this approach, it is also convenient to make trade-offs between processor performance and flexibility of the architecture. Furthermore, this approach can be applied efficiently for architectural design space exploration.

Nevertheless, in order to be retargetable, a compiler has to be machine independent to a certain extent. The compiler can be adapted to a certain target machine by writing custom machine-specific components or by providing a model of the target machine. It seems that retargetability inherently tends to compromise code efficiency. This is due to the fact that the fewer assumptions the compiler

can make about the target machine, the less machine-specific hardware features can be exploited to generate efficient code.

1.2.3 Post-pass optimization

Once assembly code has been generated, a significant optimization potential is still left, which should take into account the concrete architecture specifications, e.g. memory architecture and address generation hardware.

- Memory access optimization. Several DSPs, e.g. Motorola 56k, Analog Devices 210x and AMS Gepard show two memory banks, which are accessible in parallel. This raises the problem of partitioning the program variables into two groups such that potential parallelism is maximized. Traditional compilers use only one of the two banks, or leave the partition decisions to the programmer by means of C language annotations. Dedicated optimization phases [69, 56, 65] are based on pre-scheduled assembly code. In [33], a memory-aware approach is proposed to adapt the scheduling decisions to the concrete memory modules. Detailed knowledge of the memory interface as well as fast access modes are both exploited in combination with the processor's pipeline timing information for scheduling.

- Address code optimization. Dedicated address generation units (AGUs) are used for enhancing the execution speed of embedded processors. Such AGUs generally contain address register files and modify register files. Efficient auto-increment(decrement) address computation is supported. Exploration of auto-increment(decrement) modes depends on the layout of variables in memory, and a large amount of techniques for address code optimization [48, 46, 68, 79] are available, which differ in concrete AGU configurations and optimization methods.

1.3 Reconfigurability

Reconfigurable instruction-set processors (RISPs) are the kind of processors that have the capability to adapt their instructions sets to the application being executed through a reconfiguration in their hardware. There is a large variation of RISP architectures, which can be roughly classified as closely-coupled and loosely coupled reconfiguration according to the integration degree with the programmable processors, coarse-grain and fine-grain reconfiguration according to the granularity of the reconfiguration hardware, etc. Much literature considers the reconfigurable hardware as being implemented on a field-programmable logic (FPL). In this case, granularity also refers to the size of the reconfigurable logic.

The building blocks for fine-grained logic are gates, which are efficient for bit manipulation operations, while coarse-grained blocks are bigger and suitable for bit parallel operations. In this thesis we move the reconfigurable part to the instruction sets, while the data path is kept wide as in some VLIW processors. The corresponding instructions to implement certain applications on the hardware can be stream (block) based instructions or custom instructions and the instruction encoding for operands can be hardwired, flexible or fixed depending on the size and performance requirements. In this work, reconfigurability is considered from a new perspective: there is a rich set of functional resources in the data path. The communication network connectivity from the data path to the register files is considered to be fixed while the instruction set is reconfigurable through a flexible instruction decoder. In this way, instruction width can be reduced for the purpose of small code size, while certain regularity in the instruction set and data path is maintained for efficient compilation.

1.4 The aim of this work

In this work, we focus on the back-end of the retargetable compilation of application specific embedded processors. Code generation is hampered by the combination of tight timing constraints imposed by signal processing applications, and resource constraints implied by the processor architecture. Additionally, highly-encoded instruction sets designed to have small code size imposes additional limits to the parallelism. Distributed register files with limited connectivity make it difficult to compile. Register file size requirements are of extreme importance for a compiler since any valid schedule must fit in the available number of registers of the target machine. If not, values have to be spilled to the background memory and this is detrimental to the code quality, since extra load and store operations have to be inserted. A phase-by-phase approach is obviously not ideal for efficient code generation. All these characteristics require a unified approach. This indicates that if we can build a uniform environment for modeling all the constraints, such as instruction set, resource and register file constraints imposed by a the target processor, then we have the potential to build a unified code generation. The static resource model (SRM) approach in this work is developed to build up such an environment.

Figure 1.11 gives an overview of the thesis. After the front-end processing, we obtain the immediate representation, i.e. the data flow graph. The timing and resource constraints are extracted from the application. At the same time, the instruction set constraints are extracted from the instruction set architecture of a processor through constructing the SRM. Thus a kind of machine description language is obtained. Together with other constraints, they form the input of FACTS,

which is our code generation and synthesis tools based on the constraint analysis techniques and several search strategies. Besides this application, the obtained SRM can also be used for ISA design to obtain a balance between code size and performance by modifying the constraints iteratively. The SRM concept is also applied to the limited address range architecture for the purpose of reducing the encoding cost of operands in instructions.

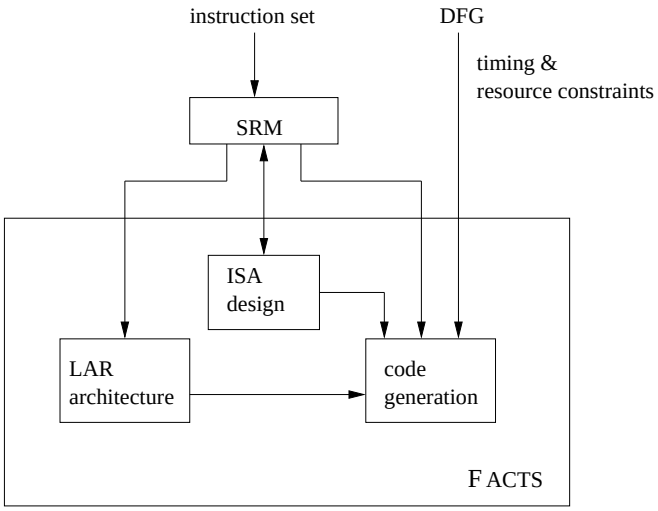


Figure 1.11: Overview of the thesis

1.5 Contributions

The work in this thesis can be summarized as follows:

- The methodology of the static resource model (SRM) is proposed for modeling the highly-encoded instruction sets of embedded processors. These constraints are combined with the resource and timing constraints such that a unified code generation approach is obtained and the phase coupling problem is overcome.

- The SRM approach is applied to a reconfigurable instruction set processor. The reconfigurable part is the instruction decoder. The instruction set constraints can be modeled efficiently through constructing the SRM for different configurations.

- The SRM approach also presents an efficient solution for instruction set design since all the constraints originated from instruction sets are represented as static resources, which can be used for the fast estimation of performance. Any modification of these resources has a direct effect on the performance and code size.

- The SRM approach is extended to cope with the limited address range (LAR) register file architecture. In this architecture, although a functional unit is physically fully connected to a register file, it reads from and writes to only a subset of registers in a register file, consequently reducing the number of encoding bits. Different functional units may access different subsets of a register file, and these subsets may overlap to maintain the communication between those functional units. The LAR architecture may cause an additional assignment problem for values to be addressed in different ranges, which results in the need for phase coupling. In this work, the assignment decisions are postponed by integrating the limiting address constraints with the value lifetime conflicts, resulting in a uniform conflict graph for register allocation.

1.6 Thesis outline

The organization of the thesis is as follows. Chapter 2 introduces the basic definitions and concepts for the development of this work. Basic block applications are represented as data flow graphs. Resource and timing constraints are administrated by the so-called "distance matrix". Constraint analysis techniques are the basis for an integrated scheduling and register binding in code generation.

Chapter 3 briefly reviews a large set of reconfigurable instruction set processors and proposes a new way of perceiving the processor reconfigurability for the purpose of efficient compilation for these processors.

In Chapter 4 the static resource model concept is explained for an efficient modeling of instruction set constraints based on the "convex hull" approach. These constraints resemble resource constraints in the sense that any combination below the upper bound is valid. Therefore constraint analysis can be adapted easily for integrated code generation. Phase coupling of code selection, scheduling and register binding is relieved. Another advantage of this method is that instruction set design can be performed quickly by tuning the instruction set constraints to balance the performance and code size, which is discussed in Chapter 5.

This methodology can also be applied to the LAR register file architectures. Chapter 6 shows this application by modifying the associated conflict graphs by including all the architectural restrictions, yielding a "modified" conflict graphs.

Therefore, the traditional coloring algorithms for register allocation can be adapted with little effort to solve the register file assignment problem.

At last, Chapter 7 concludes the contributions of this work and discusses the future research topics.

Chapter 2

Constraint Analysis

Code generation methods for digital signal processors are increasingly hampered by the combination of tight timing constraints imposed by signal processing applications and resource constraints implied by the processor architecture. Traditional methods often separate code generation into several phases. This results in sub-optimality (or even infeasibility) of the generated code because it ignores the problem of phase coupling. This increases the necessity for automated techniques that can cope with different kinds of constraints during scheduling. In this chapter, we will introduce the constraint analysis techniques which can deal with the tight combination of timing constraints and resource constraints. By exploiting the constraints to prune the schedule search space, the scheduler is prevented from making decisions which violate those constraints.

2.1 Data flow graph

In this section, we will introduce the basic concepts shared in architectural synthesis and code generation.

2.1.1 Data flow graph

In the analysis-synthesis model of a compiler, the front-end translates a source program into an intermediate representation from which the back-end generates the target code. Often the intermediate representation is expressed as graph model and is partitioned into *basic blocks*.

A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end.

Each basic block is represented by a data flow graph (DFG), which describes the primitive operations performed in that block, and the dependencies between them.

Definition 2.1 (Data flow graph) *A data flow graph DFG = (V, E, W) is a directed, edge-weighted graph, where*

- V *is the set of vertices(operations),*
- $E = E_d \cup E_s$ *is the set of precedence edges,*
- $E_d \in V \times V$ *is the set of data precedence edges(values),*
- $E_s \in V \times V$ *is the set of sequence precedence edges, and*
- $w : E \rightarrow \mathbb{Z}$ *is the function describing the time delay associated with each precedence edge in clock cycles.*

A DFG describes the primitive operations performed in an algorithm and the precedence edges define a partial order on the execution of the operations. The *execution delay* of an operation is the number of clock cycles needed for the completion of the operation and is captured as attribute to the precedence edges. Two vertices (dummy operations) are always assumed to be present in the DFG: the *source* and the *sink*. They have no execution delay, and represent the relative execution of the first operation and the last operation. Often they are not shown when depicting a DFG. Operations can have an execution delay of multiple cycles and can be pipelined with a data introduction interval (or restart time when the functional unit is ready for reuse), which are modeled using precedence constraints [53]. They can have multiple inputs and multiple outputs. In this case, it is assumed that the target architecture provides the corresponding connection from the functional units to the register files.

A data edge $u \in E_d$ represents a *value*, which is produced only once and may be consumed several times. After compilation, it has a unique lifetime and is assigned to one storage unit. A *variable* is also associated with one storage unit. It can consist of one or more values produced and consumed with non-conflicting lifetimes. Because using values helps to reduce the complexity in the implementation of DFG, the value approach, e.g. the single static assignment approach is used and a value renaming or variable unfolding [43] [76] is assumed to be performed for each variable prior to any analysis.

An example DFG is shown in Figure 2.1. In this figure, operations are $source$, $n0$, $n1$, $n2$, $n3$, $n4$, $n5$, and $sink$. The corresponding execution delay for these operations are 0, 1, 1, 2, 1, 1, 1, 0. Data dependency edges are a, b, c, d, e. They are drawn with solid lines and are all with weight 1. Sequential edges are

drawn with dashed lines. Weights for sequential edges are shown explicitly. For example, $w(source, n0) = 0$, $w(source, n1) = 0$, $w(n1, n4) = 3$, $w(n3, n5) = 3$, $w(sink, source) = -6$.

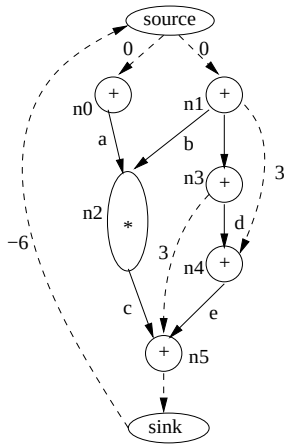


Figure 2.1: A data flow graph

2.1.2 Timing constraints

Timing constraints refers to the execution of operations in an application that must meet requirements in terms of time, e.g. start time, total time. The task of *scheduling* is to assign each operation $o \in V$ a start time $s(o)$. Start times are constrained by the precedences. A precedence edge $(o_i, o_j) \in E_d \cup E_s$ states that:

$$s(o_j) \geq s(o_i) + w(o_i, o_j) \tag{2.1}$$

A chain of precedence edges $o_i \rightarrow o_k \rightarrow \cdots \rightarrow o_j$ with total added weights d is called a *path*, implying that $s(o_j) \geq s(o_i) + d$.

Definition 2.2 (Distance) *The distance $d(o_i, o_j)$ is the length of the longest path from operation o_i to o_j.*

A path in the DFG thus represents a minimum timing delay between two operations. These distances are stored in a *distance matrix*, which administrates the length of the longest path between every pair of operations.

Other timing constraints can also be expressed as distances and stored in the distance matrix.

Latency: The number of available cycles for scheduling a DFG is defined as the *latency* L. It is expressed as a sequence edge from the sink to the source with the weight $-L$. According to inequality 2.1, it is interpreted as $s(source) \geq s(sink) - L$, which is equivalent to $s(sink) \leq s(source) + L$. Because the source is always scheduled in clock cycle 0, this formula implies that the sink should be scheduled in clock cycle L or earlier. Since all other operations precede sink, it indicates that all of them have to finish their execution within L clock cycles.

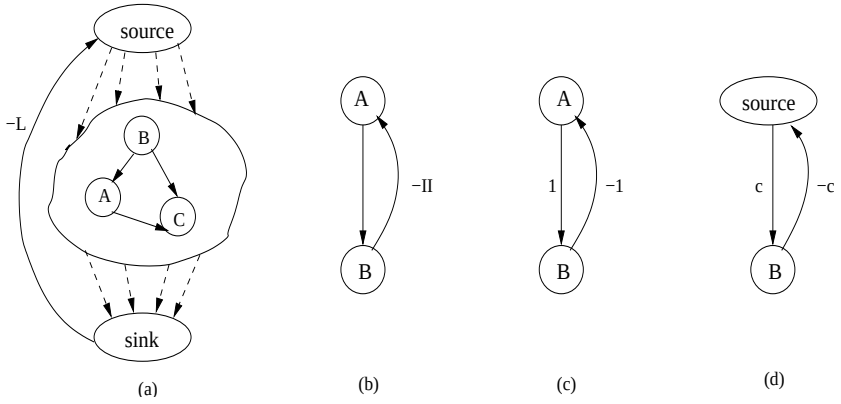


Figure 2.2: Modeling a constraint on (a) the latency L (b) the initiation interval II (c) pipelined execution and multicycle operations (d) scheduling decision

Initiation Interval: *Loop pipelining* or *loop folding* is the technique to enhance the performance of applications on architectures with high levels of parallelism. Unlike the schedules in which one iteration of a loop is executed strictly after the execution of the previous one, pipelined schedules overlap multiple iterations. Iterations are periodically initiated with a period called *initiation interval II*. If values are assigned to registers, it has to be ensured that each value belonging to one iteration has to be consumed before it is overwritten by the production of the same value in the next iteration. This means that a value cannot be alive longer that II clock cycles. Thus this constraint is modeled by a precedence edge with weight $-\mathrm{II}$ from the consumer to the producer operations of a value, which is illustrated in Figure 2.2 (b).

Pipelined executions and multicycle operations: Pipelined executions and multicycle operations can be modeled by introducing an operation for each stage of the execution. Subsequent stages are linked in time using two sequence edges as indicated in Figure 2.2 (c). For multicycle operations, A and B occupy the same resource.

Scheduling decision: Scheduling decisions may take different forms. A timing relation between two operations can be directly translated to a sequence edge. When an operation B is fixed at a certain clock cycle c, we need two sequence edges as indicated in Figure 2.2 (d) to fix the start time of B.

2.1.3 Resource and storage constraints

Resource conflicts are modeled by introducing the concept of *resources* and by defining the *resource usage* for each operation. In a processor architecture, a *functional resource* can be used in many ways. E.g., a functional resource ALU can execute an operation add or a sub, etc. For reasons of complexity, we do not wish to enumerate all possible uses of a functional resource. Therefore we consider the collection of these uses and associate with it an *operation type*. In this example, operations add and sub are associated with the operation type alu. Let T denote the set of operation types. Then the function $\tau : V \rightarrow T$ defined the operation type for each operation. Each operation type is associated with a resource type, which is characterized by a *delay value*, a *data introduction interval* for pipelined resources and the *number of instances* available for each resource type.

The delay value is the number of clock cycles that a functional resource takes to accomplish its task. Pipelined resources are the functional resources that perform operations on different data sets concurrently, therefore they consume and produce data at time intervals that are smaller than the execution delay. The data introduction interval is the number of clock cycles that a functional resource takes to be ready for reuse with a new set of inputs operands.

Resource constraints can be used to model more abstract constraints, like those arising form an instruction set. Consider the case that no instruction exists for the parallel execution of operations o_i and o_j, this can be modeled by generating an *artificial* resource [73] with only one instance, that is "used" by both o_i and o_j. Thus the constraint analysis techniques, introduced in Section 2.3, will point out that o_i and o_j cannot be scheduled in the same clock cycle since there is only one (artificial) resource available for executing o_i and o_j each time.

Storage constraints are presented using *memory types*. Each memory type is characterized in terms of its access type (random-access register file, fifo, stack, rotating register file), the number of available units, and the number of elements per unit in case of fifos and stacks.

2.2 Schedule search space

In order to obtain the scheduling result for an application, it is convenient to describe the set of possible solutions, the *solution space*. The solution space is the range of possible start times for each operation, which can be approximated by the ASAP-ALAP schedule intervals of one operation based on the analysis of precedence constraints.

Sequencing constraints are added explicitly to the DFG as precedence constraints, yielding a reduction in the ASAP-ALAP intervals. In this way, a more accurate estimation of the set of feasible start times is obtained.

2.2.1 Apparent schedule space

In order to measure the effect of the additional precedence constraints on the schedule freedom of operations, the "apparent freedom" or *mobility* of operations is defined as the average difference between the ALAP and the ASAP start times of operations $o \in V$:

$$mobility(DFG) = \frac{\sum_{o \in V} (ALAP(o) - ASAP(o))}{|V|} \tag{2.2}$$

Because the performance of a scheduler depends largely on the accuracy of the ASAP-ALAP interval estimation, mobility before and after the constraint analysis is used as the performance measure of the analysis.

2.2.2 Distance matrix

Although the interval representation is relatively simple, it is not an entirely accurate representation for ordering (precedence) information. Thus we resort to the *distance matrix*.

The distance matrix captures the relative timing: the minimum and maximum difference between the start times of each pair of operations in the data flow graph. The distance matrix is calculated using an all-pairs longest-path algorithm, an adaptation of the all-pairs shortest-path algorithm presented in [18].

Timing constraints can be expressed directly in the distance matrix. Resource constraints are associated with module execution intervals. Each one is the timing interval for certain resource to execute an operation. These execution intervals are in fact restrictions on timing. Storage constraints are analyzed using the constraint analysis techniques [53], and the results are expressed as precedence constraints, which can be easily integrated with the distance matrix by updating the distances of pairs of operations in the data flow graph: incrementing the minimum distance

or decrementing the maximum distance between two operations. Thus the effects of all the constraints can be stored in the distance matrix.

An obvious drawback of maintaining the distance matrix is that it is computationally expensive. However, constraint analysis results in a set of sequence edges and since adding an extra sequence edge requires only incremental update instead of recalculating the whole distance matrix, the run time is acceptable in practice [75].

2.3 Constraint analysis techniques

The basic constraint analysis techniques consists of *rules* that are triggered by different types of constraints and produces a set of sequence edges. These sequence edges are used to prune the schedule search space to prevent the scheduler from making wrong decisions to cause infeasible solutions.

2.3.1 Execution interval analysis

The execution interval analysis method analyzes resource constraints by examining the intervals in which operations can be executed [72] [73]. It reduces the execution interval of an operation when observing that no resource is available for executing that operation in the corresponding clock cycles. This is illustrated in Figure 2.3.

A data flow graph is given in Figure 2.3 (a). The latency L is assumed to be 5 clock cycles, and one resource type 'adder' is available for executing all the operations. The execution intervals are determined by the [ASAP,ALAP] values as depicted in Figure 2.3 (b). They are called initial *operation execution intervals* (OEIs).

From the available resources, the so-called *module execution intervals* (MEIs) are calculated. Each MEI represents the abstract notion that some resource has to execute an operation.

Execution interval analysis combines the execution intervals of the operations with resource constraints by constructing a *bipartite schedule graph* (BSG) in the following way: the operations and their corresponding OEIs are put on the left side. The MEIs are shown on the right side. They are always ordered according to their start and end times. There is an edge between an OEI and a MEI if the intervals overlap, indicating that the corresponding operation can be executed in the designated MEI. In addition, this requires that all preceding operations can be matched with the preceding MEIs. The same holds for succeeding operations.

The key observation of the analysis is that for every feasible schedule, there exists a *complete matching* in the bipartite schedule graph between the OEIs and

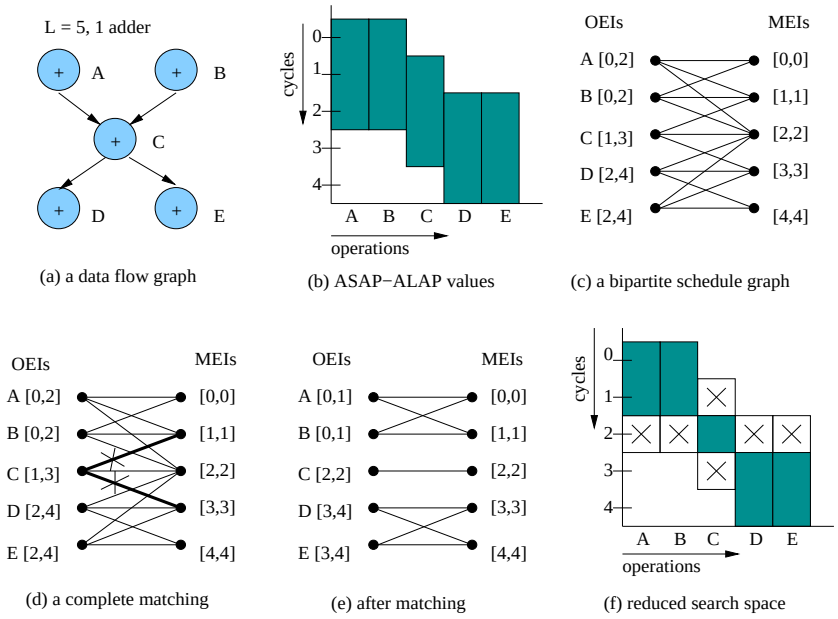


Figure 2.3: Execution interval analysis

the MEIs. That is, every OEI is matched to exactly one MEI and vice versa. This analysis uses the algorithm of [66] to identify edges that can never be part of a complete matching, and these edges are removed from the bipartite schedule graph. In this example, operation C can only be executed in MEI [2,2]. This adjustment corresponds to a pruning of the search space and is depicted in Figure 2.3 (f).

Execution interval analysis contains an additional analysis: to determine the earliest possible start time of an operation, a relaxed scheduling problem involving all its predecessors is solved to determine the lower bound of the first clock cycle. The scheduling problem is relaxed in a sense that the precedence constraints are essentially ignored. It is only enforced that each operation cannot start earlier than the lower bound of its start time. Similarly, all successors of the operation are analyzed. The predecessors and successors are determined using the distance matrix: An operation o_i is said to precede an operation o_j iff $d(o_i, o_j) \geq 1$.

2.3.2 Loop folding analysis

Basic constraint analysis essentially contains several *rules* [52] considering the timing relation between conflicting operations in case of loop folding. It is based

on the fact that two conflicting operations cannot be scheduled in the same *potential*. Potential is the time slot where operations in different loops are scheduled together. The potential associated to a time t is $t \ mod \ II$. So if two operations o_i and o_j have a resource conflict and the distance between these operations would cause them to be scheduled at the same potential, i.e. $d(o_i, o_j) \ mod \ II = 0$, the distance has to be increased by at least one clock cycle. This is depicted in Figure 2.4.

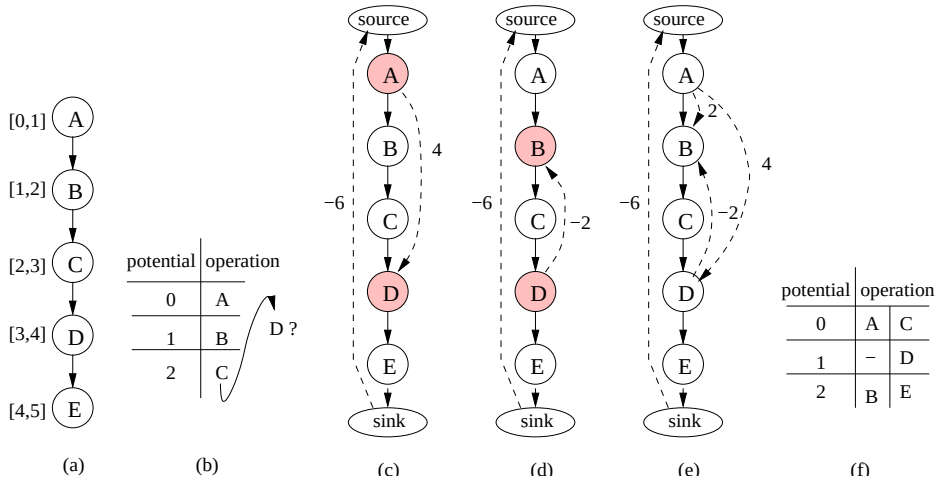


Figure 2.4: Resource conflicts result in precedence edge in loops: (a) a loop kernel with initiation interval 3, latency 6, resource conflict A and D, B and D (b) list scheduling reports infeasibility (c) conflict between A and D results in sequencial edge with weight 3+1=4 (d) conflict bwteeen B and D results in sequencial edge with weight -3+1=-2 (e) the two sequential edges cause $A \rightarrow B$ increased by one

In Figure 2.4 (a), a data flow graph of five operations is given. It is assumed that the initiation interval is 3, latency is 6. Resource A has a conflict with D, and the same holds for resources B and D. The [ASAP, ALAP] interval is printed left to each operation. In order to meet the constraint of three clock cycles on the II and six clock cycles on the latency, loop folding has to be applied. The possible result of list scheduling is shown in Figure 2.4 (b). The left column contains the *time potential* (schedule time t modulo II). The list scheduler greedily schedules A, B and C as soon as possible, and concludes that D cannot be scheduled, while below we prove that a feasible schedule exists by applying the rules of our constraint analysis.

In Figure 2.4 (c), (d), (e) sequentially, we observe a path $A \rightarrow B \rightarrow C \rightarrow D$ with a length of 3. Because $3 \ mod \ II = 0$, we can add a sequence edge from A

to D with weight 3 + 1 = 4 because of the resource conflict between A and D. Similarly, there is a path $D \rightarrow E \rightarrow sink \rightarrow source \rightarrow A \rightarrow B$ of length -3 clock cycles. Because of the resource conflict between D and B, this length has to be increased to -3 + 1 = -2. The effect of combining the two analysis results is visible in the distance matrix by computing the longest paths induced by the individual sequence edges. As a result, the distance between A and B is updated to 2 clock cycles, corresponding to the final feasible schedule in Figure 2.4 (f) where B is postponed with one clock cycle.

2.3.3 Storage constraint analysis

Storage constraint analysis arises from the limited availability of storage resources. In this section, we introduce relatively simple cases, e.g. two values have to be assigned to the same register, illustrated in Figure 2.5. More detailed rules for storage constraint analysis have been studied in [52]. When two values are assigned to the same register, their corresponding lifetimes are forced to be serialized. In general, this can be done in two ways: value u precedes value v or vice versa. Sometimes there already exists a precedence between the various accesses to u and v that excludes one of these possibilities. This can be seen from Figure 2.5.

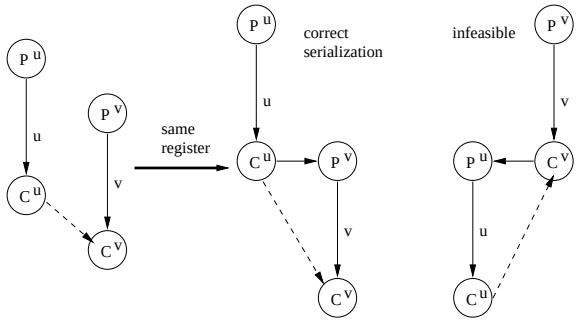


Figure 2.5: To solve the register conflict, C^u has to precede P^v

In this situation, because there already exists a precedence $C^u \rightarrow C^v$, the sequence edge $C^u \rightarrow P^v$ is a necessary and sufficient constraint to solve the register conflict between u and v.

2.4 Conflict graphs and coloring

Conflict graphs are used for storage file capacity satisfaction and allocation. In this section, we introduce the basic preliminaries and related definitions.

Definition 2.3 (Conflict Graph) *A conflict graph $CG(RF) = (V^c, E^c)$ is an undirected graph, where*

- *V^c is the set of vertices representing values to be allocated to a storage file, and*
- *$E^c \subset V^c \times V^c$ is a set of edges. There is an edge $(u, v) \in E^c$ if there is a conflict between u and v.*

The *degree* of a vertex is the number of edges incident to it.

A *subgraph* of a graph CG is a graph of which the vertex and edge sets are contained in the vertex and edge sets of CG. A *clique* is a subset of vertices that induces a subgraph of CG in which those vertices are all pairwise. The *clique number* $\gamma(\mathrm{CG})$ is the number of vertices of the maximum clique of the CG.

Vertex coloring of a graph consists of assigning a color to every vertex such that no two adjancent vertices have the same color. *Exact coloring* refers to the coloring using the minimum number of colors. The *chromatic number* $\chi(\mathrm{CG})$ is the smallest possible number of colors required for coloring the CG.

A color corresponds to a register binding decision. For application specific instruction set processors, usually there is a limited connection between the functional units and the storage files. So a value cannot be assigned to each register. Consequently, not all the colors can be exploited when assigning a color to a vertex in the conflict graph. We introduced the Annotated Conflict Graph (ACG) definition.

Definition 2.4 *An Annotated Conflict Graph(ACG) is an undirected graph represented by a tuple $(V^c, E^c, c, \mathbb{Z})$, where*

- *V^c is the set of vertices,*
- *$E^c \subset V^c \times V^c$ is a set of edges denoting conflicts, and*
- *$\mathbb{Z}$ is a set of colors.*
- *The mapping $c(v) \rightarrow 2^{\mathbb{Z}}$ defines the "color set" for each $v \in V^c$.*

2.5 Unified code generation

As we mentioned in Chapter 1.2.1, there are strong dependencies between the phases of code generation of application specific processors. If a decision is made in certain phase, consequently it has an impact on the other phases. Thus phase coupling has to be relieved in order to obtain efficient code for these processors. Constraint analysis has the ability of dealing with the integrated resource and timing constraints with the help of the above mentioned techniques. Furthermore, the distance matrix provides the basis of a unified administration of the search space. Therefore potentially constraint analysis has the ability for a unified code generation if all the architectural constraints of a processor and timing constraints of an application can be described and combined.

2.6 Research tool FACTS

FACTS is a research tool developed to take advantage of timing and resource constraints to prune the schedule search space. By exploiting these constraints, the scheduler is often prevented from making a decision that inevitably violates one or more constraints.

The structure of FACTS consists of three layers [75], as depicted in Figure 2.6. The core layer contains the internal representation of the algorithm to be scheduled and the scheduling search space. At the intermediate layer, the basic constraint analysis techniques are provided. On top of that, search strategies are implemented.

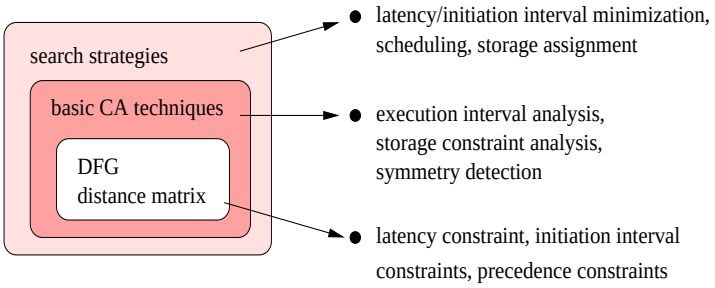


Figure 2.6: The layer structure of FACTS

At the core layer of FACTS, each basic block of the algorithm to be scheduled is represented by a data flow graph. In addition, the schedule search space is represented by a distance matrix. This distance matrix administrates the minimum

and maximum difference between the start times of each pair of operations in a DFG.

The results of the constraint analysis techniques in FACTS are conceptually expressed as additional sequence constraints in the DFG. The effects of these results on the schedule search space are combined and computed by updating the longest paths between each pairs of operations.

The essence of constraint analysis is that additional sequence edges are added that are necessarily implied by the combination of other constraints. In this way, the schedule search space is pruned to prevent the scheduler from finding infeasible solutions, without eliminating feasible solutions.

Search strategies are dealt with at the top level. The objective may be to minimize some criterion like the latency or the initiation interval, or to satisfy more global constraints like a fixed capacity of a register. These constraints are global in the sense that they have a general effect on *all* timing relations, without affecting any *specific* timing relation.

Chapter 3

Reconfigurable Instruction Set Processors

3.1 Introduction

Reconfigurable processors are processors with architectures that can be reorganized in different ways for different applications. The reconfigurable part can be a coprocessor, a functional unit in the data path or an ISA. In most cases the reconfigurable part is mapped on re-programmable hardware logic. In this thesis, we mainly focus on two categories: reconfigurable data path processors based on field-programmable logic (FPL), and reconfigurable instruction set processors. In the first class, hardware components, usually FPL based on FPGAs, are added to the core processor and correspondingly, the instruction sets are tuned towards this architecture adaptation. Usually there is a reconfiguration delay for the FPL, which has an impact on the performance by increasing the total execution delay. In addition, it adds the necessity for hardware/software partitioning, which is not convenient for the compiler. In the second class, the data path contains normal units, just like RISC processors or VLIW processors, while the instruction set can be tuned towards different applications through a reconfigurable instruction decoder. It has the advantage that the reconfiguration timing overhead is kept low and the conventional compiler techniques can still be used to generate efficient codes for different applications.

This chapter is organized as follows: Section 3.2 discusses the related work in the two classes of processors. In Section 3.3 the proposed architecture template is given with the discussion on area estimation and compiler effort.

3.2 Related work

In this section, we briefly review previous work on the above mentioned two categories of reconfigurable processors. A lot of research has been done on the first type of processor, while in this thesis we focus on the second type and provide an architecture template. The compiler adaptations, especially the phase coupling problem of code generation for such kind of architectures is discussed in the next chapter.

3.2.1 Coarse-grain and fine-grain reconfiguration

A useful criterion to classify reconfigurable processors is the granularity of the FPL itself. Depending on the degree of reconfigurability it possesses, the FPL can be classified as: fine-grained FPL, which is configurable at the level of individual bits, and coarse-grained FPL, which is configurable at the level of individual words.

One convenient way of synchronizing data communication is to embed a coarse-grain hardware block in the data path of a processor. The instructions control the data traffic to the hardware block via its registers. This reduces the memory space required to buffer data traffic while waiting for the access to the bus in the co-processor architecture.

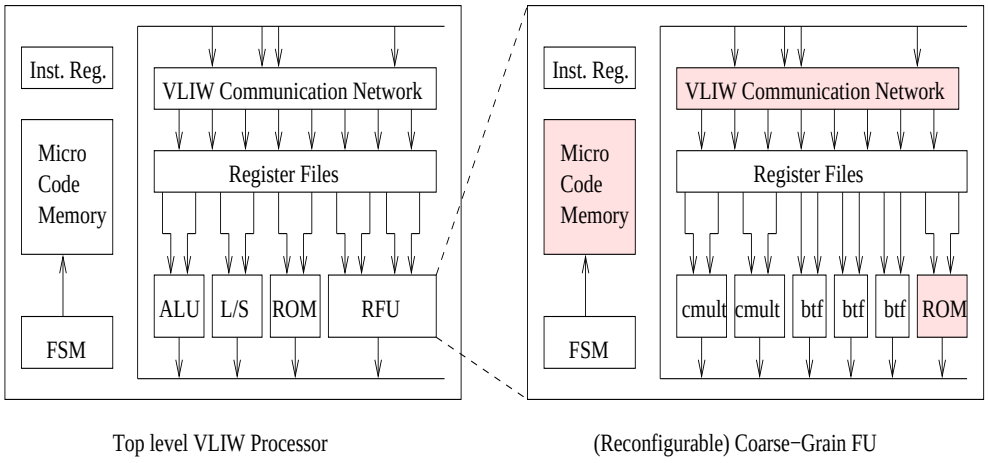


Figure 3.1: A VLIW data path with a coarse-grain application-specific unit

Busa et al. [14] proposed a synthesizable VLIW architecture with a coarse-grain functional unit which takes the form of a VLIW processor itself, see Figure

3.1. A substantial speed-up can be achieved as well as a reduction in code size. In addition, the input and output operands to and from the coarse-grain unit can be individually controlled to obtain advantageous signal lifetimes, thereby reducing the pressure on the data path registers.

The initial schedule of the coarse-grain FU will be partially taken into account while scheduling the application. In this way, a FU's internal schedule could be considered as embedded in the application's schedule. The example in Figure 3.2 depicts the scheduling of a coarse-grain FU. A small application is implemented in the coarse-grain FU in order to achieve a limited instruction width. Its DFG with the I/O operations is shown in Figure 3.2 (a). The embedding processor controls only the timing of the I/O operations. Therefore the original DFG can be simplified to a single coarse-grain operation in Figure 3.2 (b). The flexibility available during scheduling I/O operations depends on the hold-ability of the coarse-grain FU. If the coarse-grain FU can be held (frozen), then the I/O operations can be further "stretched" away from each other, as in Figure 3.2 (c), to provide or withdraw data in a "just in time" fashion. If the coarse-grain FU cannot be held, the timing relations between the I/O operations are fixed, modeled by the additional sequence edges in Figure 3.2 (d).

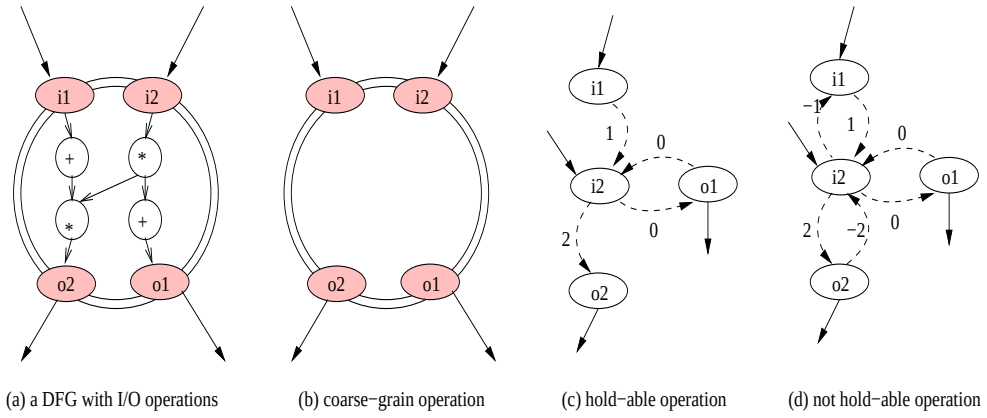


Figure 3.2: An embedded coarse-grain operation

In Figure 3.3, we compare the execution length and code size for a set of architectures with and without an application specific unit (ASU). The ASU can be one functional unit or a set of functional units. It can also be a reconfigurable unit (RC-ASU).

In the given example, we consider an application containing two critical loops. Each loop requires a different configuration for efficient acceleration. The RISC-

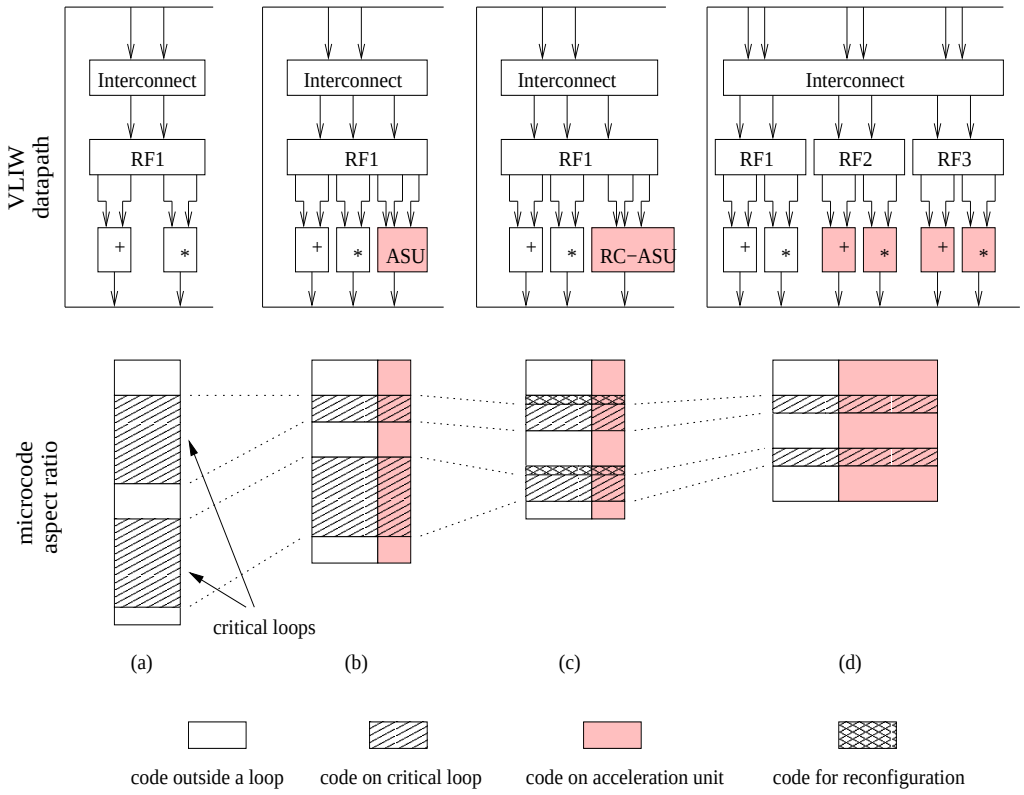


Figure 3.3: The compared VLIW data paths, and their relative controller's microcodes, as compiled for an application containing two different DSP loops (a) a RISC-like data path (b) a fixed coarse-grain VLIW data path (c) a reconfigurable coarse-grain VLIW data path (d) a "flat" VLIW data path

like architecture in Figure 3.3 (a) does not provide any hardware acceleration at all. The microcode width is very small, but the execution of both critical loops is long, with consequently a delay in terms of performance and an expansion in terms of code size. On the contrary, a "flat" VLIW processor, which contains many fine-grain FUs, will accelerate both loops at the cost of large instruction width , as shown in Figure 3.3 (d). A fixed coarse-grain VLIW processor in Figure 3.3 (b) will provide hardware support for the acceleration of one of the two loops, while keeping the microcode width moderately small. Finally, a reconfigurable coarse-grain VLIW processor in Figure 3.3 (c) will accelerate both loops with relatively small code size.

The embedded hierarchical VLIW processor in Figure 3.3 (b) contains a local internal microcode for the execution of the internal schedule of the coarse-grain operation. For a typical coarse-grain operation such as an 8-point DCT, the microcode is about 20 instructions by 25 bits. Note that the "flat" processor in Figure 3.3 (d) requires several 100 instructions by 25 bits to control the added resources. Most of these bits represent NOPs because the added resources are only used in the critical loops.

In case of reconfigurable coarse-grain units, the configuration latency of the internal microcode varies from 2 to 10 cycles. The controller microcode as well as the RTL VHDL for the RC-ASU have been generated using the architectural synthesis tool A|RT Designer [8].

3.2.2 Closely-coupled and loosely-coupled reconfiguration

According to the integration degree of the FPL with the programmable processor, two categories can be defined: closely-coupled reconfigurable processor or loosely-coupled reconfigurable processor. In the closely-coupled processor, the FPL is integrated within the data path of the host processor and is directly controlled by instructions issued in the host processor. In the loosely-coupled processor, the FPL is integrated as a co-processor of the host and is activated by run-time scheduling. Communication with the host is performed via the system buses.

Loosely-coupled reconfiguration

The advantage of loosely-coupled approach is that it is relatively easy to integrate dedicated hardware in a plug-in-play like fashion. The disadvantage is that the system bus is burdened with heavy data traffic from and to the application specific block. This reduces the predictability and the performance of the system.

In a reconfigurable context, this block usually consists of an FPGA. The number of bits required to configure the block is in the order of 500k bits, and

typically takes a few milliseconds to configure at a clock frequency of 50-100 MHz.

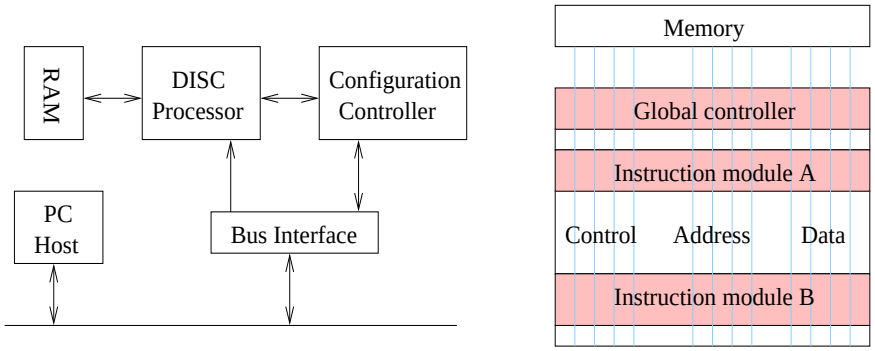
For example, PRISM is a fine-grained, loosely-coupled reconfigurable processor. In the PRISM-I [9] prototype, a standard programmable processor (Motorola 68010 running at 10MHz) is augmented with an FPGA board containing four Xilinx 3090 devices. Both processor and FPGA board are connected to a 16-bit system bus. The compiler partitions the application into software and hardware, executed in the processor and in the FPGA board respectively. A list of functions that can be implemented in hardware is prompted and it is the programmer who makes the partition decisions. A limitation of PRISM is the communication latency between processor and FPGA accelerator. In addition, large chunks of the application must be mapped onto the FPGA, which possibly does not fit well with the FPL architecture.

Closely-coupled reconfiguration

Closely-coupled approaches can be further divided into Reconfigurable Functional Units (RFUs) and Reconfigurable Data path Segments (RDSs) according to the degree of integration in the data path. In the first case, an RFU is added in the execution stage of the pipeline. In the second, more than one stage of the pipeline can be by-passed, and the control flow diverted to the added FPL resources.

- DISC [81] is a fined-grained, closely-coupled RDS processor operated at ISA level, which supports demand-driven modifications of the instruction set. In DISC, the instructions are implemented as instruction modules, which are individually configured on the partially reconfigurable FPL resources as demanded by the application program. This kind of dynamic instruction paging removes idle instruction modules and reduces configuration time considerably. In addition, the system state can be saved on the FPL during configuration. The ability to partially configure custom instruction modules allows DISC to implement an important strategy–relocatable hardware. DISC implements relocatable hardware in the form of a linear hardware model, which consists of a uniform communication network and a global controller. The global controller specifies the communication protocol, controls global resources and monitors circuit execution. The communication network provides access to global resources for all instruction modules and performs intermodule communication. It is decomposed into fixed vertical buses for control, address and data.

  The DISC processor needs a host to manage the runtime reconfiguration of the instruction modules. When a new module is needed, the host evaluates the current state of the FPL and chooses a physical location for the



(a) DISC system

(b) DISC linear hardware space

Figure 3.4: DISC architecture

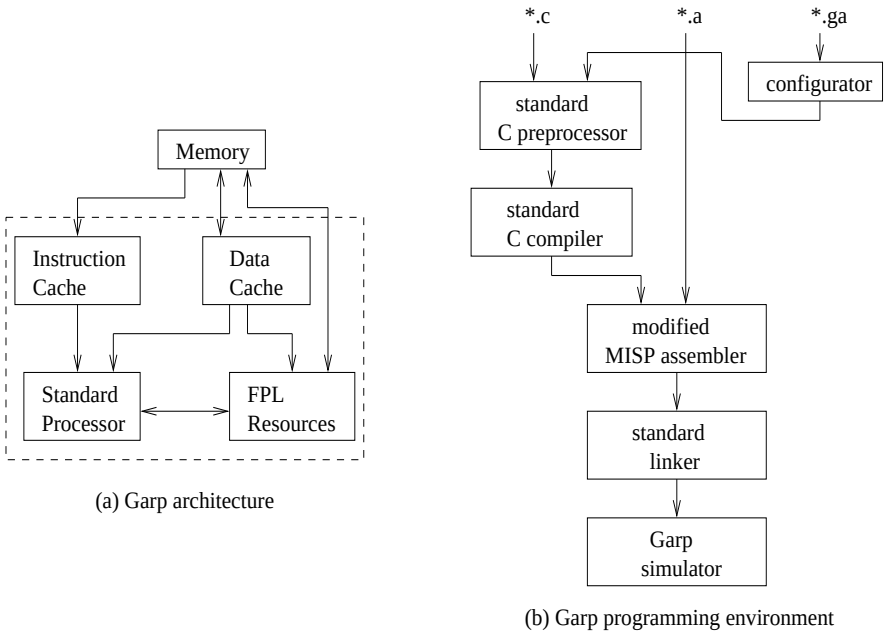
requested module. If possible, the new module is placed in a non-occupied position. Otherwise, a least-recently-used algorithm is applied to remove the idle modules. The host also relocates modules at runtime.

One drawback of partially configuring the device during runtime is the overhead caused by continually reconfiguring instruction modules. Although a set of retargetable tools is available, no automatic hardware/software partitioning and synthesis is implemented. Instruction modules are hand-crafted and stored in a library. Because the global controller implements only simple operations, the FPL needs to implement most computations, even those that could be efficiently implemented on a hard-wired ALU. Consequently, it cannot be optimized for a specific kind of applications.

- Garp [38], as DISC, adopts a closely-coupled RDS approach. It is based on a MIPS processor instead of a global controller. The loading and execution of configurations on the reconfigurable array is under the control of a program running on the main processor. Several instructions have been added to the MIPS-II instruction set for this purpose.

  Garp makes external storage accessible to the reconfigurable array through the standard memory hierarchy. Distributed within the array is a cache for recently used configurations, and programs can quickly switch between several configurations without the cost of reloading from the memory each time. It resembles DISC in the way that function modules are mapped onto the FPL in horizontal rows and global buses are orthogonally located through the rows. Four memory buses run vertically through the rows for

moving the information into and out of the array. For moving data between array blocks, orthogonal wires with various length are available. The logic blocks in the array are configurable at the granularity of a pair of bits. This decision is made based on the assumption that most configurations taken up by multiple-bit operations are configured identically for each bit. In this way, the size of configurations and timing for loading configurations are greatly reduced, which compromises the critical computation performance and flexibility.



(a) Garp architecture

(b) Garp programming environment

Figure 3.5: Garp architecture and compilation

Garp utilizes traditional VLIW compilation techniques in the context of automatic hardware/software partitioning. The partitioning strategy is that the standard MIPS data path is used for control, system interfacing and other non-critical tasks. The FPL array maps the relatively complex application segments that can read and write data directly to memory, such as the entire loop bodies. In addition, loop bodies mapped onto the FPL can be pipelined.

- PRISC [63] is the first to use the RFU approach. In PRISC, a small fined-grained FPGA-based RFU is inserted into the execution stage of a standard RISC pipeline, in parallel with the standard functional units, as shown in

Figure 3.6 . The RFU is stateless, so that no FPGA state has to be saved with a context switch. The RFU must also execute in a single clock cycle to prevent synchronization difficulties in the pipeline. The reconfiguration control logic, typically a finite state machine, is responsible for reading configuration data from memory/bus and controlling the sequencing loading.

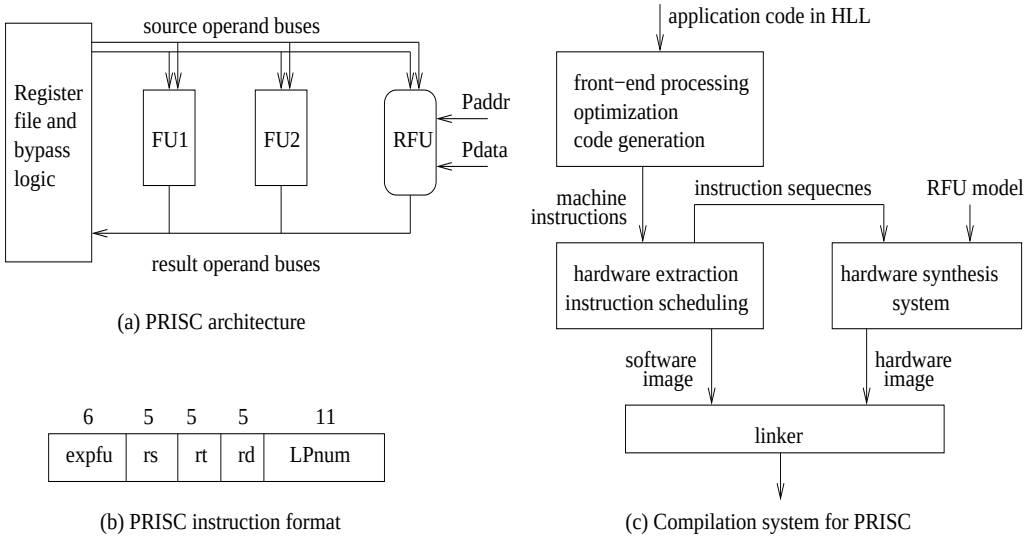


Figure 3.6: PRISC architecture, instruction format and compilation

To program and operate RFU, a new 32-bit instruction *expfu* is defined to evaluate a boolean function with two inputs and one output, see Figure 3.6 (b). The *LPnum* field specifies the particular boolean function to be executed which is extracted from the application. A 11-bit register *Pnum* is associated with the RFU. It contains the logic function currently programmed into the physical RFU. If the LPnum in the instruction matches the value in the Pnum register, the expfu instruction executes normally. Otherwise, an exception is raised.

In the PRISC compilation system as depicted in Figure 3.6 (c), the application in a HLL is parsed, optimized and translated into the target machine instructions. These instructions are assembled and scheduled to produce a binary executable. At the same time, hardware extraction through profiling is performed to identify sequential instructions which could potentially be implemented on the RFU. Logic synthesis takes the input function and outputs a netlist of look-up tables (LUTs). Placement and routing is run

to determine if the LUT netlist fits in the resources offered by the physical RFU. The result is fed back to the hardware extraction iteratively. In the compiler of PRISC, hardware partitions are now as small as a short sequence of instructions. Typical examples are conditional constructs and bit-level parallelism. Only one operation at a time can be configured in the RFU. Every time a different RFU operation is needed, the pipeline is stalled while the RFU is reconfigured, which limits the ability to accelerate the application's core loops with multiple RFU operations. It also has the limitation that only two source operands are available for RFU operation.

- Chimaera [37] uses a special approach for RFU to overcome the limitation of reading only two operands. It's architecture is shown in Figure 3.7. The FPL has direct access to individual bits of a subset of eight registers. The address of these registers need not to be encoded in the instruction word because the FPL is already hard-wired to the desired input bits during configuration. This eliminates the register addressing flexibility of an individual RFU operation.

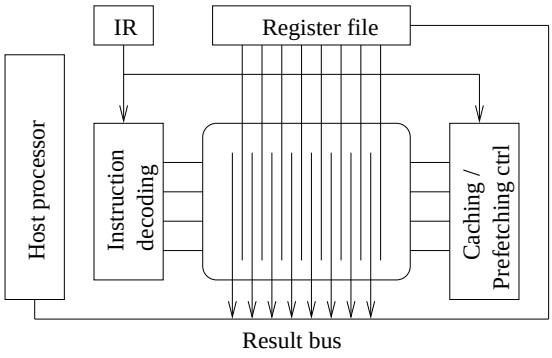


Figure 3.7: Chimaera architecture

The general partitioning strategy of Chimaera is similar to that of PRISC. RFU operation must be executed in one cycle after being scheduled. The FPL architecture is stateless and is tuned to irregular bit manipulation. Chimaera's RFU may contain several custom operations at a time. Similar to DISC, different operations occupy different rows in the FPL array. Thus, cross-optimization between operations during logic synthesis, placement and routing is not possible.

- In [42], ConCISe [42] is proposed to integrate a reconfigurable hardware acceleration unit based on CPLD in the data path of a RISC microproces-

sor, as depicted in Figure 3.8. An RFU is added to the execution stage of a standard RISC pipeline, receiving the same two source operands from the register file as the ALU and it must be executed in a single clock cycle. Unlike PRISC, more than one custom operation can be configured in the RFU concurrently. This is obtained through compiler-time reconfiguration. All custom operations that make up the hardware partition of any given application are known at compiler-time, and they are all encoded together into a single RFU configuration. Therefore, there is one RFU configuration per application program. The corresponding instruction, the so-called program-specific instruction (PSI), is encoded as a MIPS register-register operation plus a 4-bit immediate filed called DEC. DEC identifies the particular custom operation within the RFU configuration that is to be executed. This encoding also has the limitation: only operations with two input operands and one output operand can be accelerated by the RFU. This requirement matches particularly well with bit-level operations in cryptology. Several algorithms are proposed to automatically identify suitable targets for hardware acceleration, resulting the speed-up of 10-60 % on the complete application for several DSP benchmarks. The RFU has a configuration memory in the order of 5 kbits, so reconfiguration time can be neglected.

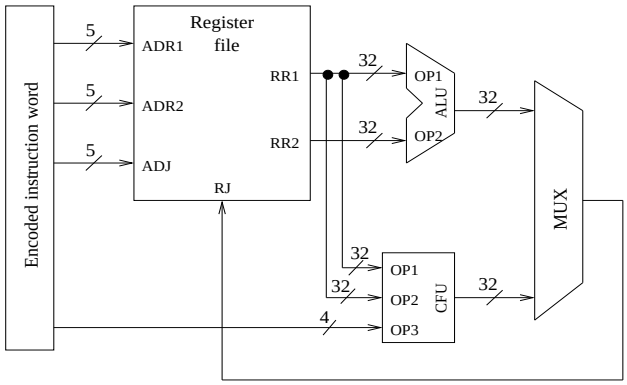


Figure 3.8: ConCISe architecture

3.2.3 Non-FPL based reconfigurable instruction set processors

In the non-FPL based reconfigurable instruction set processors, the instruction set architecture can be (re)configured such that for different applications or different segments of one application, the most suitable and code-size efficient instructions

are used for the application(s) to be implemented on the processor. This has the advantage that all the functional unit's resources are fully utilized and no extra hardware cost is added. The focus is on configuring different instruction set architectures while still make it easy for the compilers to generate efficient code.

- Tensilica's Xtensa [70] is a 32-bit configurable and extensible processor. Xtensa's processor generator provides user selectable parameters from a wide range of configurable options. The instruction set architecture can be configured with optional functional units, different sizes and types of memory interface, optional debug modules and/or and user defined instructions. Using the instruction extension language (TIE), the designer can create his/her own instructions and have them immediately understood by the C/C++ compiler, debugger and instruction-set simulator. A subset of verilog is used to describe the new function, which is built into the hardware. The profiler information helps the user to identify the bottleneck of the program, select the new configuration options or add new TIE instructions, which provides a more efficient architectural exploration.

- The REAL [83] architecture, as illustrated in Figure 3.9 contains a data path with two independent 16x16 bit multipliers, four parallel 16-bit ALUs which can be combined into two 40-bit ALUs (including eight overflow bits each), and a number of parallel shifters and saturators. In addition to the arithmetic units there are two address calculation units (ACUs) and two data memories. Effective processing performance can be enhanced by introducing application specific execution units (AXUs). AXUs are defined by the customer and can be placed anywhere in the data path or the address calculation units. In combination with a tuned instruction set a substantial reduction in cycle count can be achieved as compared to conventional general purpose DSPs.

  The full power of the data path is mainly used within the inner loop of a typical application, and much less power is needed in the "administrative" code around such loops. A distinction is made between the "standard instruction set" of the 16 and 32 bit opcodes, where only a limited parallelism is possible, and "application specific instructions" (ASIs), which allow the full parallelism being exploited. The ASI concept allows up to 256 VLIW instructions in a lookup table inside the REAL DSP being controlled by 96 bits. These are triggered by a special class of 16 bit instructions, stored in the normal program memory. The ASI lookup table can be a RAM (for prototype chips), ROM, a synthesized netlist or a combination of them. The REAL instruction set is very efficient with regard to code size, but the irregular instruction set and the data path are not convenient compiler targets.

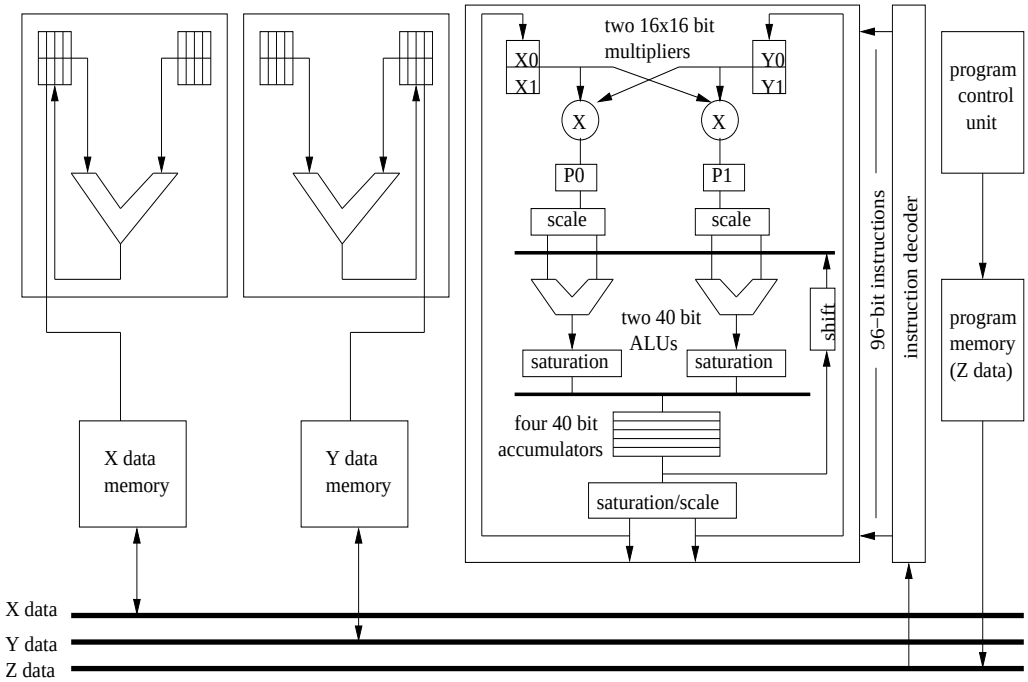


Figure 3.9: REAL architecture

3.3 Proposed architecture

The REAL architecture, treated in the previous section, exploits, among others, the fact that different parts of the application require different amounts of parallelism. In the more control dominated (sequential) parts, the instruction set is not burdened with the control of the complete parallelism offered by the data path, thereby saving program memory. The performance on the inner DSP loops is obtained at the cost of wide instructions (ASIs). In this section, we will refine this idea by introducing a reconfigurable instruction decoder and a method for run-time configuring the instruction set based on the performance required individually in each of the inner DSP loops [54]. Furthermore, by an appropriate modeling of the resulting instruction set, the approach is supported by any conventional resource-constrained (e.g. list) scheduling in order to exploit the parallelism offered by the instruction set.

Figure 3.10 and Figure 3.11 depict the relevant data path elements and the instruction decoder respectively. The communication and the register infrastructure is not shown since they are beyond the scope of this discussion. The purpose of the instruction decoder is to efficiently control the (possibly large) number of functional units in the data path. An instruction word consists of a header followed by a number of issue slots, e.g. IS1, IS2, ... in Figure 3.10. One issue slot controls the execution of an operation on a functional unit. The header bits control the demultiplexers at each of the issue slots, so they determine which exact functional unit is controlled by a certain issue slot. Issue slot i is connected (via the demultiplexer) to a limited cluster CL_i of functional units in order to restrain the area and delay complexity and to maintain scalability of the architecture. Two or more clusters may overlap to ensure sufficient flexibility in configuring the instruction set. In the example architecture in Figure 3.10, each issue slot is connected to 8 functional units. The demultiplexer accounts for 200 gates. In order to restrict the number of header bits, the control of the demultiplexer associated with slot i is limited to a subset SL_{ij} of the available functional units in CL_i. This subset is determined by the configuration of the instruction decoder, and is a way of tuning the instruction set to the application. In the example architecture in Figure 3.10, each issue slot controls, depending on the configuration, any 4 out of the 8 functional units connected to the issue slot. In the example instruction decoder in Figure 3.11, the second stage decoder (60 gates) holds this configuration and expands the (4x) 2 bits from the first decode stage to the (4x) 3 bits to control the demultiplexer. The first decode stage generates these (4x) 2 bits from the 5 header bits in the example, according to the reconfiguration look-up table. In the example, this look-up table has a cost of about 600 gates. Each decode stage corresponds to a pipeline stage. The first and most complex decode stage is optional to allow trade-off between the instruction width and the decoding delay.

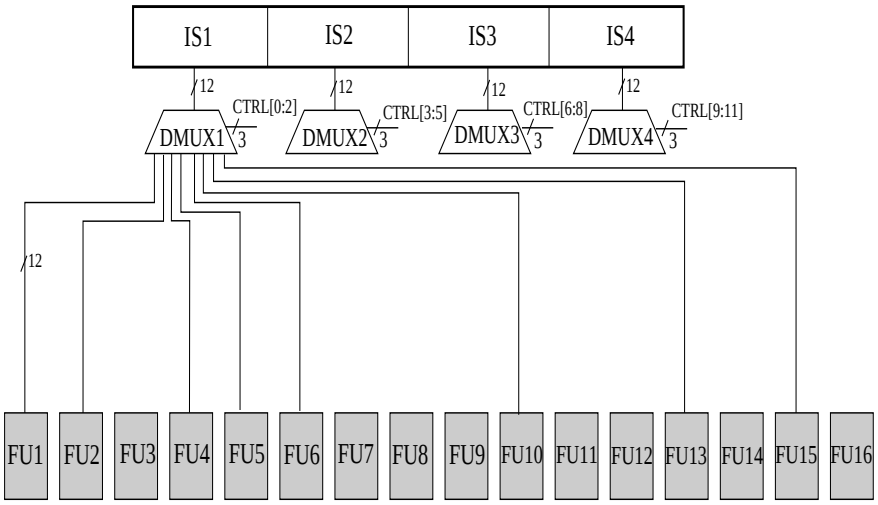


Figure 3.10: Proposed data path

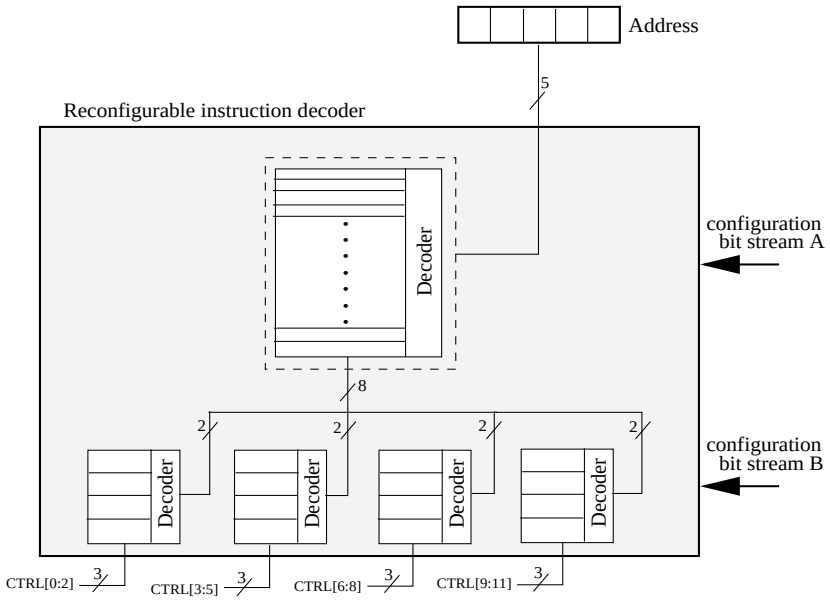


Figure 3.11: Proposed instruction decoder

Chapter 4

Static Resource Models of Instruction Sets

4.1 Introduction

The combined issues of performance requirements for meeting real-time constraints on the one hand, and code size requirements on the other hand, have caused the instruction sets to be highly encoded and to exhibit an irregular structure. The main issue introduced by an irregular instruction set is the issue of *phase coupling*: on one hand, if instruction selection is performed prior to scheduling, the optimal schedule can easily be eliminated as a result of the choices. On the other hand, if scheduling is performed first, the available instructions may not be able to implement the schedule. Traditional methods perform these tasks in different phases, thereby yielding inferior schedules.

In this chapter, we present a new approach that eliminates the need for explicit instruction selection by transferring the constraints from the instruction set to static resource constraints. All resulting schedules are then guaranteed to correspond to a valid implementation.

The outline of this chapter is as follows. Section 4.2 discusses the related work in instruction selection and phase coupling in code generation. Section 4.3 gives the definition of static resource model and the related preliminaries of convex hull and Section 4.4 presents the problem statement and our approach. In Section 4.5 we show the construction of the static resource model for both orthogonal instruction sets and more general cases. Section 4.6 presents the experimental results for a variety of instruction set architectures.

4.2 Related work

Related work in code selection and phase coupling varies a lot and depends heavily on the target architecture. Many code selection approaches for application specific processors originate from code selection for general purpose processors. Phase coupling problem for these processors is focused for generating efficient code. Section 4.2.1 reviews tree covering techniques exploited in general purpose processors and the adaptations for DSPs and ASIPs. Section 4.2.2 discusses the phase coupling in code generation for these processors and Section 4.2.3 reviews the approaches for solving the phase coupling problem statically.

4.2.1 Tree covering techniques for instruction selection

The phase of instruction selection has received a lot of attention in the software compiler community. During instruction selection, the operations in the algorithmic model are bound to register transfers or partial instructions, supported by the target processor's instruction set. Traditional compilers use the *template pattern base* to represent the target processor, which essentially enumerates the different partial instructions available in the instruction set. Each partial instruction is represented as a pattern, usually a data flow tree (DFT), which is expressed by means of the intermediate representation (IR) of the algorithm. In this pattern, nodes correspond to variables, constants, and operations, while edges denote data dependencies. It has been shown that instruction selection is an NP-complete problem for IRs that take the form of a directed acyclic graph [31]. However, optimal vertical code can be generated in polynomial time, when the following conditions are satisfied:

1. the intermediate representation of the algorithm is an *expression tree*;
2. the template pattern base is restricted to contain only *tree patterns*, i.e. it can be represented as a *regular tree grammar (RTG)*;
3. the processor has a *homogeneous* register structure, in which all the registers are interchangeable.

Several code-selector generators are based on a stepwise partitioning of the instruction selection problem, using *dynamic programming*. It is assumed that conditions 1) and 2) are satisfied. Tree pattern matching is done in a bottom-up transversal of the subject tree in [2]. For each node, the method computes the minimal cost to cover the subtrees rooted at that node. During the top-down transversal of the subject tree, the tree cover is finally found, by determining the minimum

cost at the tree's root node. A number of tools are available for automatic generation of tree pattern matchers from instruction set grammar specifications, such as Twig [1], Beg [24], Iburg [29] and Olive [1]. Since several DSPs have special registers which are used for certain functional units, dynamic programming is also extended to target heterogeneous register structures. CBC [27] is based on the hardware description language nML and the machine specification is transformed to an Iburg specification. In [7], the data flow graphs (DFGs) are pruned to become data flow trees (DFTs), which satisfy the RTG criterion and code selectors are generated by the code selector generator Olive.

4.2.2 Phase coupling in code generation

Besides instruction selection, the task of machine code generation comprises register allocation and instruction scheduling. Traditionally, these phases, each of which is an NP-hard optimization problem in itself, are solved sequentially and heuristically. However, there is a cyclic dependence, since each of the three phases may impose possibly unnecessary and obstructive restrictions on the remaining ones. All these are *phase coupling* problems in embedded code generation. Early techniques of data routing incorporate register allocation for distributed register files and instruction scheduling [64, 36, 78].

In [6], the tree pattern matching technique has been coupled with register allocation and scheduling for a family of DSPs from Texas Instruments. Later, additional heuristics for handling data flow graphs (DFGs) are presented in [7], in which the DFGs are pruned to become DFTs in order to provide faster instruction selection. An improved method without pruning based on simulated annealing has been described in [45]. Mutation scheduling [55] is another approach to considering phase coupling, where algebraic transformations are exploited in order to explore alternative instruction set mappings. A constraint logic programming technique for DSPs with irregular architectures has been presented in [12]. In that approach, all binding decisions are delayed until they are really required, which yields a maximum degree of freedom in code generation. However, this is at the expense of high compilation time.

Another approach for instruction selection is based on constructing graphs which include the complete information of the architecture. In CHESS [77], the target processor is described in the nML language[27] and is translated into an Instruction-Set-Graph (ISG), which models connectivity, encoding restrictions and structural hazards. Instruction selection covers the control-data flow graph with partial instructions (bundles) by searching valid paths in the ISG. All these methods add a complex step to the compiler chain and eliminate potentially interesting solutions from the schedule and register binding search spaces.

Exact approaches for code generation are described in the form of constraints

(generally linear equations and inequalities). The complete solution space is explored while all constraints are considered simultaneously, leading to a complete phase integration. The approach in [47] performed instruction scheduling based on Integer Programming (IP) approach. Code selection and register allocation are performed in advance, based on Iburg generated code selectors. Scheduling is delayed. Complete integration is given in [80, 32]. Wilson's approach [80] leads to very large runtime due to large IP models. The approach of Gebotys describes constraints based on Horn clauses [32]. This can be mapped to Linear Programming (LP) problems. However, only restricted classes of architectures can be handled efficiently by these approaches. Architectures comprising heterogeneous register files typically lead to an explosion of generated models. In [35], a covering approach for DFGs for finding minimum set of (VLIW) instructions is specified. The approach is based on binate covering. Detailed register allocation is performed in a post processing phase.

4.2.3 Solving phase coupling problem statically

A completely different approach to tackle the code generation problem is to exploit the instruction set constraints statically. State diagrams or Finite State Automata (FSA) are used to represent the set of all legal instruction schedules for a processor [11, 58]. They provide the advantage of space and time efficiency. However, they are not amenable to certain advanced scheduling techniques, such as iterative modulo scheduling [60] and mutation scheduling [55]. The concept of reservation tables is used to detect conflicts for scheduling. Examples of compilers that adopt this approach include the Multiflow Trace Scheduling Compiler [49] and the Trimaran (Elcor) Compiler [34]. Eisenbeis [23] formalized the conflicts among functional units within one issue slot by using reservation tables for VLIW architectures, such as the Trimedia processor. Timmer et al. [73] discuss the modeling of instruction set constraints together with resource constraints for instruction scheduling, but the assumption that all the instruction set constraints can be transferred to resource constraints targets mostly instruction sets with a structure associated with so called Issue-Slot machines. The virtue of these orthogonal instruction sets, like those of the VLIW paradigm [61], is that they allow to a large extent the modeling of the instruction set constraints by means of a model like [23, 73], or a static resource model. These models offer the scheduler more opportunity to satisfy the timing and resource constraints than with the use of an explicit instruction selection step. The scheduler rather than the instruction selector is considered the designated place for handling these constraints. The disadvantage of the orthogonal processors (especially VLIW processors) is the inherent problem of code size due to their associated large instruction word widths.

One of the code generation approaches based on constraint analysis is motivated by the fact that the irregularity of the architectures of ASIPs and DSPs and the timing objectives of certain tasks to be implemented on those processors can be represented as *constraints* and phase coupling can be performed by pruning the search space defined by those constrains. In this way, wrong decisions are prevented by checking whether it leads to collisions of the constraints. In [53], data flows are expressed as precedence constraints. Latencies and initiation intervals are transferred to timing constraints. Register files are constrained by their capacities, and functional resources are associated with operation types. Constraint analysis is the kernel idea to combine all those constraints. Resource constraints are translated into precedence constraints. Thus when one decision is made, its impact on the whole search space can be assessed.

The focus of this chapter is on replacing the instruction set constraints of highly encoded instruction sets by a static resource model, allowing the use of code size efficient processors in combination with efficient compilation tools. Our static resource model is not restricted to issue slot tables with fixed bit-width for each issue slot, thus it can deal with more flexible encodings. In addition, the procedure of constructing a static resource model also provides useful information for instruction set design.

4.3 Definitions

In this section, we introduce the preliminaries in the computational geometry field and the basic concept of the static resource model.

An *instruction* is a multi-set of operations which can be executed in atomically. For an operation *op* in an instruction I, we denote the number of times that it appears in I as $N_I(op)$. If for two instructions I_0 and I_1, $N_{I_0}(op)$ is always at most equal to $N_{I_1}(op)$ for each operation *op*, we say that instruction I_0 is *contained* in instruction I_1. For a set of instructions, if instruction I always contains the other instructions, then we say I is the *maximum instruction*. In this thesis, we consider instruction sets (IS) where for each instruction all contained instructions are also in IS. We call these instruction sets *prefix closed*.

Our approach is motivated by the observation that both resource constraints and instruction set constraints can be expressed as inequalities. For example, if an architecture contains two ALUs and each ALU can be used as an adder or a subtractor, then the resource constraints can be expressed as the following inequality: $N(ALU) \leq 2$. Any schedule satisfying at any time the above inequality indicates a valid resource usage. Similarly, if an instruction set contains instructions $[add, add]$, $[add, sub]$ and $[sub, sub]$, the operation usage can be expressed as an inequality: $N(add) + N(sub) \leq 2$, assuming any subinstruction is also a valid

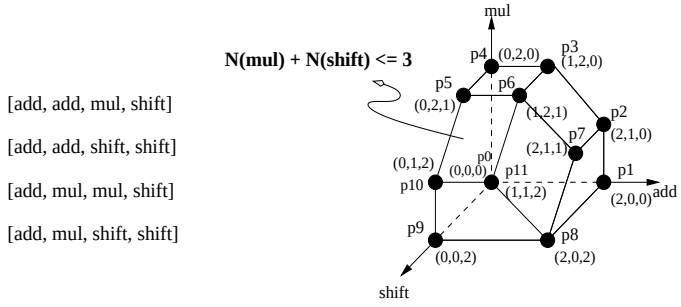
instruction.

In a processor architecture, a functional resource can be used in different ways. E.g., a functional resource ALU can execute an operation *add* or a *subtract*, etc. For reasons of complexity, we do not wish to enumerate all possible uses of a functional resource in an instruction set. Therefore, we associate multiple operations using the same functional resource with one *operation type*. We denote the set of operation types with T. Operation types are used consistently in the whole thesis.

Definition 4.1 (Static Resource Model) *A Static Resource Model (SRM) is defined by:*

- *a set of resources R,*
- *a set of operation types T,*
- *the number of instances of each resource $r \in R$, denoted by #r, and*
- *a function that associates each operation type with a multiset of resources in R that it needs*

In general, operation types are associated with axes in a multi-dimensional space $\mathcal{R}^d$, where d is the dimension corresponding to the number of operation types, and instructions can be geometrically represented as points in this operation type space. An example is depicted in Figure 4.1 (b). Figure 4.1 (a) gives an instruction set IS using three operation types. Only the maximum instructions are listed in IS; instructions that are contained in other instructions are not explicitly represented. We will do so throughout this thesis. Since the instruction set uses three operation types, the operation type space is 3-dimensional. As mentioned, all instructions correspond to points in this space. Instruction $I_1 = [add, add, mul, shift]$, for example, is drawn as point $p7$ with coordinate values $I_1(add)$, $I_1(mul)$ and $I_1(shift)$. Because we assume prefix closedness of instruction set, the instruction set corresponds to a well-defined, closed subspace of the operation type space. It is our aim to capture this subspace spanned by the instructions via inequalities. Inequality $N(mul) + N(shift) \leq 3$, for example, is one of the inequalities needed to capture this subspace. Inequalities can subsequently be translated to *virtual resources*. Some of the inequalities are directly related to functional resource constraints. Others result in additional restrictions on parallelism. In this thesis, we call the latter one “extra virtual resources”. In general, deriving the inequalities and the resulting virtual resources for an instruction set can be perceived as a *convex hull* problem [10] [57]. Here we give some preliminaries of the convex hull problem.



(a) instruction set IS (b) geometrical representation of IS

Figure 4.1: Instructions expressed as points in the operation type space

Definition 4.2 *Given a set $X = \{x_1, x_2, \cdots, x_k\}$, where $x_i \in \mathcal{R}^d, 1 \leq i \leq k$, y is called an* affine combination *of X if y can be expressed as a linear combination of X:*

$$\vec{y} = \sum_{i=1}^{k} \lambda_i \vec{x_i}, \lambda_i \in \mathcal{R}^d, 1 \leq i \leq k \qquad and \qquad \sum_{i=1}^{k} \lambda_i = 1 \tag{4.1}$$

A *convex combination* of X is an affine combination such that each λ_i is non-negative. A *proper convex* combination is one where each λ_i is positive.

Definition 4.3 *A subset S of a d-dimensional space $\mathcal{R}^d$ is called a* convex set *if and only if every convex combination of points in S is also in S. The* convex hull *of the set of points in X, denoted as* conv(X)*, is the set of all convex combinations of X; it is the smallest convex set containing X.*

Consider again Figure 4.1. The set of points bounded by the planes $mul = 0$, $add = 0$, $shift = 0$ and the other planes depicted constitute the convex hull of the original instruction set IS. That is, it is the smallest convex set containing all the instructions. (Recall that all subinstructions of the listed instructions are also included in this instruction set.) Above, it has already been explained that the depicted subspace corresponds to the instruction set. Thus, in this example, the convex hull of all instructions captures precisely all the instructions in the instruction set. Moreover, it does not contain a combination of operation types that is not a valid instruction. This property turns out to be crucial in translating instruction set constraints to resource constraints. Later in this chapter, an example is given showing that not all instruction sets have the property that they are precisely captured by their convex hull.

As the example discussed so far already suggests, a convex set can be described by means of its boundary. In general, a convex set is determined by a set of halfspaces. A halfspace is the set of all points below or above some plane. A halfspace can be seen as a *constraint* on the elements of the set being described and it is defined via an inequality. Such an inequality describes part of the boundary of the convex set. Consider the example of Figure 4.1 again. The halfspace containing all points below the plane through points $p5$, $p6$, $p10$ and $p11$ corresponds to inequality $N(mul)+N(shift) \leq 3$. A convex set can be seen as an intersection of a set of halfspaces. The convex set in the example of Figure 4.1 is the intersection of nine halfspaces, namely $N(add) \geq 0$, $N(mul) \geq 0$, $N(shift) \geq 0$, $N(add) \leq 2$, $N(mul) \leq 2$, $N(shift) \leq 2$, $N(add) + N(mul) \leq 3$, $N(mul) + N(shift) \leq 3$, and $N(add) + N(mul) + N(shift) \leq 4$. The nine inequalities define the boundary of the convex set, and thus of the convex hull of the given instruction set. An interesting observation is that any set defined as the intersection of a set of halfspaces is necessarily convex.

Given a set of halfspaces $\mathcal{H}$, halfspace $H \in \mathcal{H}$ is *non-redundant* if there is some point included in every halfsapce in $\mathcal{H} \setminus \{H\}$ but not included in $\mathcal{H}$. Thus, a non-redundant halfspace really restricts the convex set. The notion of non-redundancy is useful in determining the minimal set of inequalities describing the convex set S. The minimal halfspace representation of the convex hull of some set X is denoted by $\mathcal{H}(X)$.

It is also possible to describe a convex hull via its *extreme points*. The convex hull in Figure 4.1, for example, can be described via the set $\{p0 = (0,0,0)$, $p1 = (2,0,0)$, ..., $p11 = (1,1,2)\}$. Given a convex set S, a point in S is an extreme point if and only if it is not a $proper$ convex combination of any two points in S. The notation $\mathcal{V}(X)$ denotes the vertex description of the convex hull of some set X, consisting the set of extreme points.

There are two closely related computational problems concerning the two descriptions of the convex hull of a set X:

- The *vertex enumeration problem* is to compute $\mathcal{V}(X)$ from $\mathcal{H}(X)$.
- The *convex hull problem* is to compute $\mathcal{H}(X)$ from $\mathcal{V}(X)$.

In code generation for ASIPs and DSPs, deriving virtual resources from a given instruction set can be perceived as a variant of the convex hull problem, since the instructions form the vertex description and the virtual resources form the non-redundant halfspace description. On the other hand, in the instruction set design space exploration, one can optimize the instruction set by modifying the SRM to meet the real-time constraints, and subsequently calculating the corresponding instructions. This can be perceived as a variant of the vertex enumeration problem.

The precise complexity of the two problems is an interesting problem. The two problems are dual and are therefore of the same complexity. Two excellent references on the complexity of convex-hull related problems are [30, 10]. Recall that d is the dimension of the space in which we are operating; let n be the number of inequalities in case of the vertex enumeration problem and the number of vertices in case of the convex hull problem. It is known that the problems are efficiently solvable ($O(n\,log\,n)$) for $d \in \{2, 3\}$. For the general case, there is an algorithm of *optimal worst-case* complexity $O(n^{\lfloor d/2 \rfloor})$ [17]. Thus, one could say that for fixed dimension d, the complexity is polynomial. However, for high dimensions, the degree of the polynomial tends to become large. In our applications, the dimension of the operation type space may become quite large, although it remains to be seen how large it will be in practical examples. Fortunately, in practice hardly any input to any of the two above problems ever causes the worst-case complexity. The average complexity is usually much better [21]. The precise performance depends on the particular implementation of the algorithms. For a comparison of algorithms, the interested reader is referred to [10].

4.4 Problem statement and approach

In this section, we generalize the problem of constructing the SRM for a given instruction set and present the convex hull approach for this problem.

Problem Definition 4.1 *The general problem of instruction selection with SRM can be defined as mapping a given instruction set to an SRM such that any schedule for a DFG satisfying the resource constraints posed by the SRM corresponds to a valid instruction selection.*

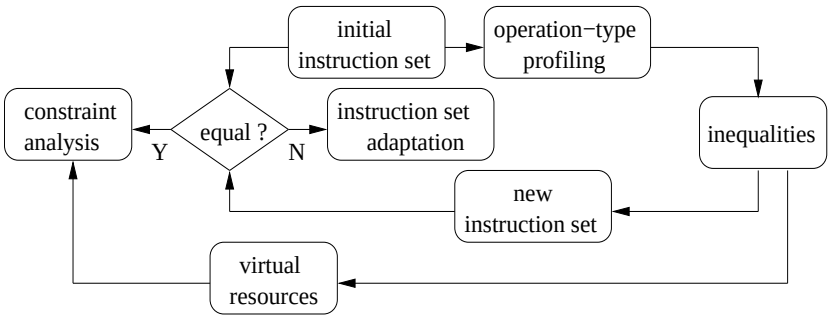


Figure 4.2: Overview of the approach

We propose a solution strategy depicted in Figure 4.2 based on expressing the instruction set constraints as inequalities, like $N(add) + N(sub) \leq 2$ in the example in Section 4.3. We start from an initial instruction set IS. It is followed by a step called *operation-type profiling*, which calculates the number of times operation types as well as their combinations appear in an instruction set. In this way we enumerate all the potential extreme points of the instruction set in the operation type space. We then search for an SRM by computing the convex hull, i.e., the smallest convex set containing all the instructions in IS. With the resulting inequalities, we can obtain a new instruction set NIS by enumerating all the instructions allowed under those inequalities. The equivalence of the instruction set constraints and the SRM is verified by comparing the new instruction set NIS to the initial instruction set IS. The SRM itself is derived directly from the inequalities, but is only accurate if the aforementioned test turns out positive. If NIS contains combinations of operation types that do not correspond to an original instruction in IS, then the constraints obtained from the SRM are not sufficiently tight. Since the convex hull is the *smallest* convex set containing all the original instructions, it even follows that it is impossible to capture the instruction set constraints via virtual resource constraints. The procedure is illustrated in Figure 4.2. In case the SRM is not accurate, it can be profitable to adapt the instruction set that it fits the SRM, because then all the advantages of the SRM approach can be exploited.

4.5 Construction of the static resource model

In this section, we first illustrate the advantage of the SRM approach in code generation with a small DFG. It is especially suitable for a resource-constrained scheduler. We compare the scheduling result with list scheduling and constraint analysis for this DFG. The construction of the SRM for orthogonal instruction sets is provided in Section 4.5.3 and the construction for general instruction sets is discussed in Section 4.5.4 and 4.5.5.

4.5.1 Advantage of the static resource model

As we mentioned before, code generation for ASIP cores makes it necessary to recognize valid instructions in the DFG. This is usually performed by covering the DFG with patterns, representing valid processor instructions, such as depicted on the left hand side of Figure 4.3.

The main issue introduced by the highly encoded instruction set is the issue of phase coupling: on one hand, if instruction selection is performed prior to scheduling, the optimal schedule can easily be missed as the result of the choices

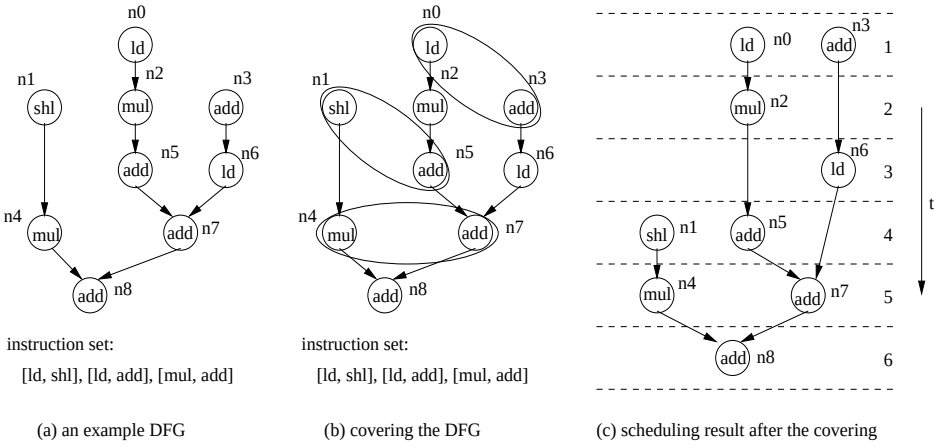


Figure 4.3: Instruction selection prior to scheduling may yield inferior results

made during instruction selection. On the other hand, if scheduling is performed first, the available instructions may not be able to meet the scheduling constraints. Traditional methods perform the tasks in different phases, thereby yielding inferior schedules. This is depicted in Figure 4.3. The DFG in Figure 4.3 (a) has been covered with machine instructions, shown in Figure 4.3 (b). The associated schedule (6 clock cycles) for this selection of instructions is given in Figure 4.3 (c). It is suboptimal because the covering decisions can not take into account which instruction is more critical for the application.

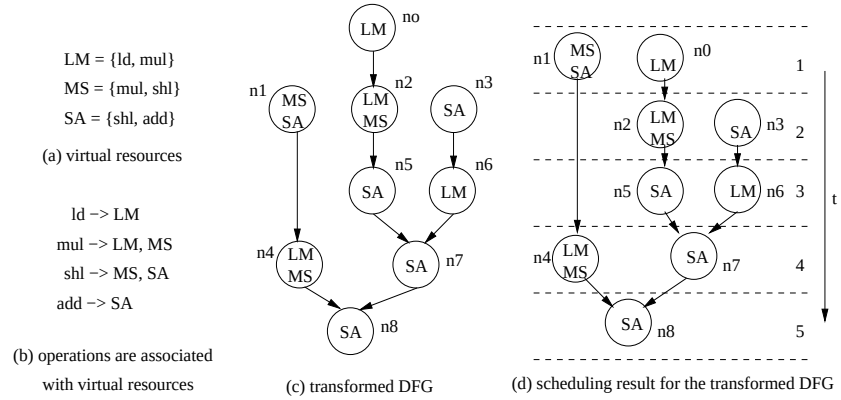


Figure 4.4: With a static resource model optimal results can be obtained

The merit of the SRM of instruction sets is that by transferring the instruction set constraints to static resource constraints, explicit instruction selection is avoided and the scheduler has the opportunity to generate an improved schedules in terms of timing and register requirements.

We illustrate our procedure in Figure 4.4. Our initial instruction set is $[ld, shl]$, $[ld, add]$, $[mul, add]$. There is no instruction available to execute operations ld and mul together in the same cycle. This conflict can be modeled as a virtual resource LM in Figure 4.4 (a) and operations ld and mul will both compete for this virtual resource. Similarly, MS and SA capture the conflicts implied by the missing instructions. The number of instances of each virtual resource is one. In addition, each operation uses the virtual resources that it is associated with. For example, mul uses the virtual resources LM and MS. The relationship between operations and virtual resources is shown in Figure 4.4 (b). By relating the operations in the original DFG to the virtual resources, a transformed DFG is obtained, shown in Figure 4.4 (c). By applying the resulting SRM of the instruction set to a resource constrained scheduler, we obtain an optimal schedule of 5 clock cycles, as depicted in Figure 4.4 (d).

4.5.2 List scheduling and constraint analysis

The resource-constrained scheduling problem is known to be intractable. Therefore, heuristic algorithms have been searched and used. One of the most popular algorithms is list scheduling algorithm.

In list scheduling algorithm, given time step l, candidate operations are those whose predecessors have already been scheduled in time steps $t_i < l$, so that the corresponding operations are completed at step l. The unfinished operations are those that started at earlier cycles and whose execution is not finished at step l. A $priority\ list$ of the operations is used in choosing among the candidate operations, based on some heuristic urgency measure. A common priority list is to label the vertexes with weights of their longest path to the sink and to rank them in decreasing order. The most urgent operations are scheduled first.

However, this simple priority rule focuses only on the delays and doesn't consider the resource constraints. For large examples and complex resource constraints, this rule might be deceptive. Because virtual resources are used to represent instruction set constraints and normally there are more virtual resources than functional resources, this chance is even higher. For example, in Figure 4.4 (c), in the first clock cycle, the candidate operations for scheduling are $n0$, $n1$ and $n3$. Since the ALAP value of $n3$ is smaller than that of $n1$, the traditional list scheduling would select node $n3$ after scheduling $n0$ and would yield a scheduling result of 6 clock cycles. Notice that $n1$ uses more virtual resources that $n3$. A better schedule can be obtained by switching the priority to resource usages and

selecting node $n1$. In general it will be very difficult to determine the appropriate priorities.

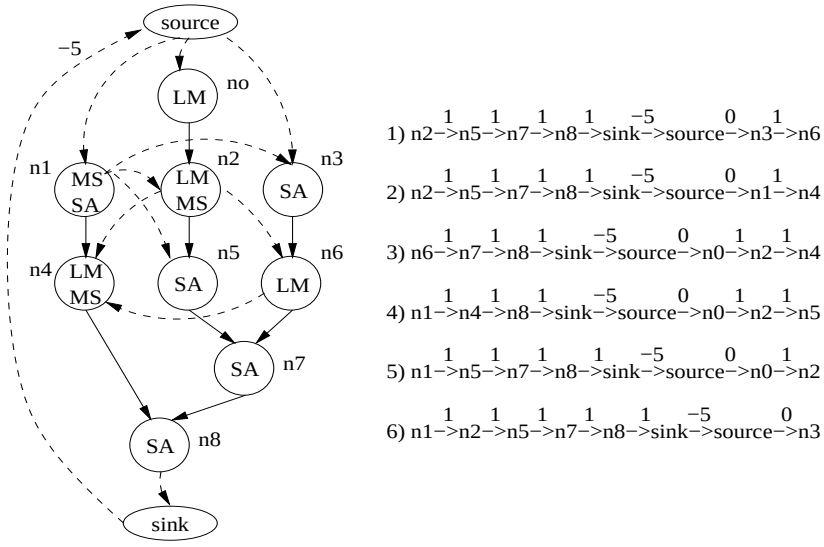


Figure 4.5: Constraint analysis for the transformed DFG

Constraint analysis can better deal with the combination of data dependencies and resource constraints as introduced in Chapter 2. The schedule search space is represented by the distance matrix. Timing constraints can be expressed directly in the distance matrix. Resource constraints are expressed as execution intervals and they are translated into restrictions on timing. Since all the constraints, i.e. data dependencies and resource constraints are stored in distance matrix and are governed by certain rules, the implication of any decision will penetrate through the whole search space. The scheduling decisions are depicted in Figure 4.5, to be explained below.

Recall the rule in Section 2.3.2: If two operations have a resource conflict and the distance between them causes them to be scheduled into the same clock cycle, then the distance has to be increased by at least one clock cycle. This rule is applied to the transformed DFG consistently and an optimal schedule is promised. As we can see, nodes $n2$ and $n6$ have a resource conflict regarding virtual resource LM. The distance from $n2$ to $n6$ calculated through path 1 in Figure 4.5 equals 0. Therefore a sequential edge with weight 1 is added from $n2$ to $n6$. For the same reason, the sequential edge from $n2$ to $n4$ is added after calculating path 2. This sequential edge will cause the distance from $n6$ to $n4$ to be increased to 0 by following path 3. Thus a sequential edge with weight 1 is added from $n6$ to $n4$.

Similarly, a sequential edge is added from $n1$ to $n5$ with yielding of path 4. This affects path 5 and results in the sequential edge from node $n1$ to node $n2$, which will further affect path 6 and a sequential edge is added from node $n1$ to node $n3$. Finally the schedule is fixed.

4.5.3 Static resource model for orthogonal instruction sets

In Sections 4.4 and 4.5, we presented an approach for deriving an SRM for a given instruction set by making use of inequalities. The basic idea is to generalize the maximum usage of operation types in an instruction set by representing it as a set of inequalities, and then transforming these inequalities into an SRM. The desired set of inequalities is derived from *operation-type profiling*. In this section, we show the usage of this approach for *orthogonal instruction sets*. An orthogonal instruction contains multiple slots for encoding multiple operations. Each slot usually encodes one operation, and the execution of the operation does not depend on the encoding of other slots. An example instruction set IS_1 is depicted in Figure 4.6 (a). The operation-type profiling is represented in the table in Figure 4.6 (b). In the table, rows correspond to the individual instructions listed in Figure 4.6 (a), and columns correspond to (combinations of) operation types. The numbers in the table indicate how many times an operation type (combination of operation types) is present in an instruction. For example, the operation type *add*, occurs twice in instruction (1), and the operation types *shift* and *mul* together occur three times in instruction (3). By looking at the largest frequency within each column, the inequalities in Figure 4.6 (c) are derived. Each inequality corresponds to one column. In general, for each combination of operation types $S = \{op_1, \ldots, op_n\}$, we have one inequality:

$$N(op_1) + \cdots + N(op_n) \leq \underset{I \in IS}{\mathrm{MAX}}\Big(\sum_{op \in I \cap S} I(op)\Big) \tag{4.2}$$

where $N(op_i)$ is the number of times an operation type appears in an instruction.

Based on the computation of the operation-type profiling, the instruction set constraints have been replaced by a set of inequalities. This set often contains redundancy, in the sense defined in Section 4.3; in the example of Figure 4.6 (c), inequality (5) can be removed because it is implied by inequality (7). Removing inequalities is advantageous because the complexity of the derived SRM is determined by the number of inequalities. From the remaining inequalities the SRM is derived, shown in Figure 4.6 (d). For example, inequality (6) is translated to the virtual resource $\{mul, shift\}$, which is abbreviated as MS, with three instances available. For reasons of convenience, we represent a virtual resource by combin-

(1) [add, add, mul, shift]
(2) [add, add, shift, shift]
(3) [add, mul, mul, shift]
(4) [add, mul, shift, shift]

(a) instruction set IS $_1$

op / I(op)	add	mul	shift	add mul	add shift	shift mul	add shift mul
(1)	2	1	1	3	3	2	4
(2)	2	0	2	2	4	2	4
(3)	1	2	1	3	2	3	4
(4)	1	1	2	2	3	3	4

(b) operation−type profiling

(1) N(add) <= 2
(2) N(mul) <= 2
(3) N(shift) <= 2
(4) N(add) + N(mul) <= 3
(5) N(add)+N(shift) <= 4
(6) N(mul) + N(shift) <= 3
(7) N(add) + N(mul) + N(shift) <= 4

(c) inequalities

#A = 2, #M = 2, #S = 2,
#AM = 3, #MS = 3, #AMS = 4

add −> A, AM, AMS
mul −> M, AM, MS, AMS
shift −> S, MS, AMS

(d) static resource model

Figure 4.6: Example of an orthogonal instruction set with its SRM

ing the first letters of all those operation types which compose it. An operation type "uses" all the resources from the SRM that it is contained in, e.g., add uses A, AM and AMS at the same time. Note that virtual resource A corresponds to the real functional resource implementing the add operation, while virtual resources AM and AMS have functionalities similar to functional resources but are not real functional resources, explaining the term "virtual resource".

The complexity of the proposed algorithm to compute an SRM for a given instruction set is $O(2^{|T|})$, where T is the set of operation types introduced in Section 4.3.

A weakness of this method is that it does not always give an SRM, even if an instruction set has one. Furthermore, a resulting SRM often has redundancies, as the above example illustrates. These redundancies are difficult to remove. The method is also not well suited for handling complex encodings. Figure 4.7 shows an example where methods from literature [23] and the one above fail to express instruction set restrictions in an SRM-like manner. The difficulty is introduced by the *wordlength constraints* on instructions. There are three adders and two multipliers in the data path. Each add is encoded with 8 bits and each mul with 10 bits. The total wordlength for an instruction is limited to 24 bits, thus all the possible combinations of operation types are: $[add, add, add]$, $[mul, add]$, $[mul, mul]$.

The method above will produce the inequalities in Figure 4.7 (b). It contains one redundant inequality: inequality (1). Furthermore inequality (3) allows the combinations of operation types such as $[add, mul, mul]$ and $[add, add, mul]$, which are in fact not valid because of the instruction encoding constraints. In contrast, the method explained below generates the inequalities for the instruction set in Figure 4.7 (c), which correctly models the instruction encoding limitation. Using this new inequality the instruction set is verified to have an SRM.

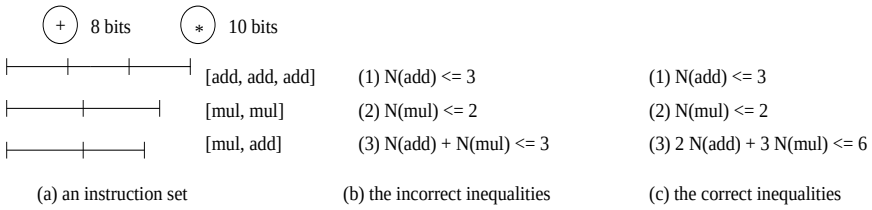


Figure 4.7: Example of an non-orthogonal instruction set and the SRM

In the next subsection, we provide a general approach for the construction of SRMs that improves the method of [86]. It solves all the mentioned problems and it gives a valid SRM for the instruction set in Figure 4.7 (a).

4.5.4 Static resource model in the general case

In order to find a method that produces a valid SRM for a broader class of instruction sets, we observe the following restriction on equation 4.2 in the approach in Section 4.5.3: The inequalities do not have weights on the left hand side for each operation type. Assume IS_1 in Figure 4.8 (a) is the full instruction set of a certain VLIW architecture, while IS_2 in Figure 4.8 (b) is a similar but smaller instruction set. Note that instruction set IS_1 is the one of Figure 4.6. The previous elementary method will generate the same SRM for the two instruction sets. Because IS_2 is smaller than IS_1, this SRM is not accurate for IS_2. However, by adding inequality (6') to the set of inequalities in Figure 4.8 (d), we solve the problem posed in the previous subsection: Instruction set IS_2 now has a valid SRM, different from the SRM corresponding to IS_1.

If the operation types are associated with the axes in a multi-dimensional space and the instructions are represented as points in this operation type space geometrically, then the inequalities derived for the instruction set can be perceived as boundaries enclosing those points and the problem is transformed into a *convex hull* problem.

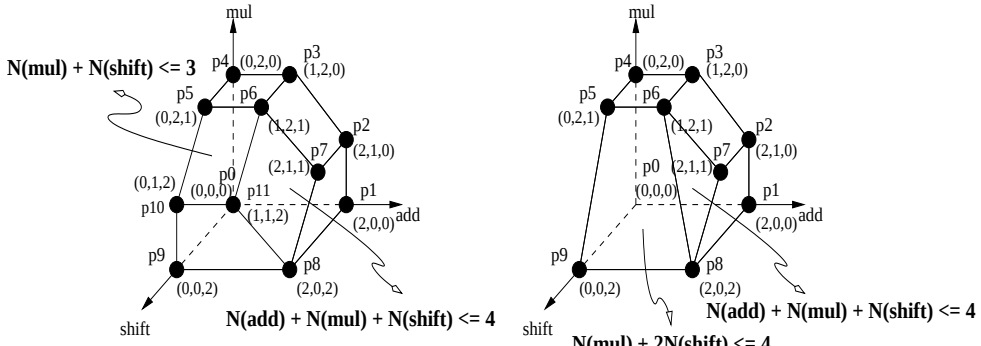
The instruction sets IS_1 and IS_2 and their convex hulls are illustrated in Figure 4.8 (a) and (b). In Figure 4.8 (c), we obtain the same set of inequalities as in Figure 4.6, which in Figure 4.8 (e) leads to the same SRM as in Figure 4.6. Notice that the algorithms used in computational geometry tools, e.g. the cdd [16] package, already omit the redundant inequality (5) in Figure 4.6 (c). Because the last instruction in IS_1 is not valid for IS_2, points $p10$ and $p11$ are not present in Figure 4.8 (b), and the convex hull and the set of inequalities in Figure 4.8 (d) are slightly different from IS_1. Notice that inequality (6'), $N(mul)+2N(shift) \leq 4$, is generated automatically with a factor greater than one for $N(shift)$. Because of this weight, inequality (3) becomes redundant, which means virtual resource S is not required any more. Also notice that for the same reason, the $shift$ operation uses virtual resource MS twice as is reflected in the SRM of Figure 4.8 (f).

The general problem of constructing the SRM for an instruction set IS can be formalized as follows: given a set of points (instructions) in the d-dimensional operation type space, determine the convex hull $conv(\mathrm{IS})$ expressed as a set of m linear inequalities. This yields the form:

$$conv(IS) = \{\vec{x} \mid A\vec{x} \leq \vec{b}\} \tag{4.3}$$

where $d = \mid T \mid$, $A \in \mathcal{R}^{m \times d}$ and $\vec{b} \in \mathcal{R}^m$.

It remains to determine how to precisely compute (4.3) for a given IS. The basic idea is to use a standard convex hull algorithm applied to the extreme points in the IS. Note that we could simply apply such an algorithm to the entire instruction set without influencing the result. However, recall from Section 4.3 that the complexity of the convex hull problem depends on the number of points n in the set of which the convex hull needs to be computed. Thus, the convex hull computation can be sped up by minimizing the input to the algorithm. Because we assume prefix closedness of instruction sets, any subinstruction is also a valid instruction. Although a subinstruction is not explicitly listed in the given example instruction sets, it is quite possible that it is an extreme point of a convex hull. For example, the four largest instructions in IS_1 can be drawn as points $p7$, $p8$, $p6$ and $p11$ in Figure 4.8 (a). These four points are not sufficient to build $conv(IS_1)$. Point $p1$, for example, represents a subinstruction $[add, add]$, which is obviously an extreme point for this convex hull. In order to construct the complete convex hull, we have to find all the extreme points in the space, which is as complex as the convex hull problem itself. A compromise between using the entire instruction set or only its extreme points for the convex hull computation follows from the following observation. As we can see in the example, extreme point $p1$ with coordinates $(2, 0, 0)$ is the projection of point $p8$ with coordinates $(2, 0, 2)$ on the plane $shift = 0$. Any point *between* them has no contribution to the convex hull, meaning it is redundant. From this intuition, we can simply calculate the points



[add, add, mul, shift]
[add, add, shift, shift]
[add, mul, mul, shift]
[add, mul, shift, shift]

(a) instruction set IS_1

[add, add, mul, shift]
[add, add, shift, shift]
[add, mul, mul, shift]

(b) instruction set IS_2

(1) N(add) <= 2
(2) N(mul) <= 2
(3) N(shift) <= 2
(4) N(add) + N(mul) <= 3

(6) N(mul) + N(shift) <= 3
(7) N(add) + N(mul) + N(shift) <= 4

(c) inequalities for IS_1

(1) N(add) <= 2
(2) N(mul) <= 2

(4) N(add) + N(mul) <= 3

(6') N(mul) + 2 N(shift) <= 4
(7) N(add) + N(mul) + N(shift) <= 4

(d) inequalities for IS_2

#A = 2, #M = 2, #S = 2
#AM = 3, #MS = 3
#AMS = 4

add −> A, AM, AMS
mul −> M, AM, MS, AMS
shift −> S, MS, AMS

(e) SRM for IS_1

#A = 2, #M = 2
#AM = 3, #MS = 4
#AMS = 4

add −> A, AM, AMS
mul −> M, AM, MS, AMS
shift −> MS, MS, AMS

(f) SRM for IS_2

Figure 4.8: SRMs created from convex hulls for IS_1 and IS_2

projected from the maximum instructions onto the planes with dimensions $d-1$, $d-2$, ..., 1 obtained by assuming 1, 2, ..., $d-1$ coordinates to be zero respectively. The complexity of this projection in the worst case is $O(k\ 2^p)$, where k is the number of maximal instructions and p is the number of operation types appearing maximally in one such maximal instruction. Of course, this approach might still result in quite a number of non-extreme points. However, we do not think that this will cause any problems in the convex hull computations. If necessary, the input to the convex hull computation may be further reduced using the techniques explained in [30].

4.5.5 Deriving resources

So far, we have explained by means of examples how instruction set constraints can be captured in an SRM based on an approach computing the convex hull of the instruction set. In this subsection, we give a theorem formulating a necessary and sufficient condition for verifying whether an instruction set has an SRM. The argument showing that the condition is sufficient includes a transformation from the inequalities describing the convex hull of the instruction set to an SRM. The basic idea of the theorem is that the convex hull of an instruction set defines a new instruction set consisting of all the integer points in the convex hull. If this new instruction set is equal to the original instruction set, then the convex hull provides the basis for an SRM of the original instruction set because it captures precisely all the instruction set constraints. If the new instruction set contains integer points that are not valid instructions in the original set, then the convex hull does not provide an appropriate SRM because the constraints are not sufficiently tight.

Theorem 4.1 *Let IS be an instruction set. If NIS is the set of all the integer points contained in $conv(IS)$, then IS has an SRM iff NIS equals IS.*

We prove the sufficiency of the condition in this theorem by giving the following SRM. Recall Definition 4.1 introducing the notion of an SRM and equation (4.3) that describes the convex hull of an instruction set in terms of inequalities. We need to define three aspects:

- the set of virtual resources R of the SRM contains all the sets of operation types corresponding to an inequality in the convex hull description of (4.3);
- each operation type needs w instances of all the virtual resources that it is contained in, where w is the weight of the operation type in the inequality in (4.3) corresponding to the virtual resource;
- the number of instances of a resource equals the bound in the corresponding inequality in (4.3).

To clarify this construction, consider the example instruction set IS_2 of Figure 4.8 (b). Figure 4.8 (d) gives the inequalities describing the convex hull of IS_2. The convex hull consists of five inequalities, which means that the derived SRM has five virtual resources. Inequality (4), for example, corresponds to virtual resource $\{add, mul\}$, as before abbreviated AM. Inequalities (1), (2), (6′) and (7) correspond to virtual resources A, M, MS and AMS, respectively. The number of instances of resource AM is determined by the right hand side of inequality (4), being 3. The other four resources have 2, 2, 4 and 4 instances, respectively. Finally, operation type mul uses one AM resource, because 1 is the weight of mul in inequality (4); it further uses one M resource, one MS resource and one AMS resource. Operation type add uses one A, one AM and one AMS resource; $shift$ uses two MS resources and one AMS resource. Figure 4.8 (f) summarizes the SRM.

It still needs to be shown that the condition in the theorem that NIS equals IS is sufficient to guarantee that the SRM defined above is an appropriate representation of the instruction set constraints of IS. The construction of the SRM guarantees that a schedule of all the operations in a DFG constrained by the SRM satisfies all the inequalities describing the convex hull of IS. Recall that NIS contains exactly all the integer points satisfying these inequalities. Thus, if NIS equals IS, the inequalities only allow valid instructions which means that any schedule constrained via the above SRM satisfies all instruction set constraints. Thus, IS has an SRM, namely the one provided above.

As an example consider again instruction set IS_2 of Figure 4.8 (b). The inequalities in Figure 4.8 (d) precisely capture all these instructions, i.e., they do not allow any integer solutions that are not instructions in IS_2. Thus, we conclude that IS_2 has an SRM. Figure 4.8 (f) provides an SRM, which is constructed as explained above.

The necessity of the condition that NIS equals IS in the above theorem immediately follows from the following fact: The convex hull of a set of points is the *smallest* convex set containing all these points (Definition 4). In other words, NIS is the smallest set of integer points containing all the instructions in IS that can be described by inequalities of the form given in (4.3). Thus, if NIS contains a point that is not an element of IS, then it is impossible to describe IS via such inequalities. Since there is a one-to-one correspondence between inequalities and SRMs, IS cannot have an SRM.

To illustrate that there are instruction sets without an SRM, consider the example in Figure 4.9. It gives an instruction set, the inequalities describing its convex hull, and the SRM that can be derived from these inequalities following the above construction. The inequalities allow instructions $[add, add, add, sub, sub]$, $[add, add, add, cmp]$, $[add, sub, sub, cmp]$, $[add, add, sub, cmp]$, $[add, cmp, cmp]$ plus all their subinstructions. Thus, unfortunately, the inequalities allow instruc-

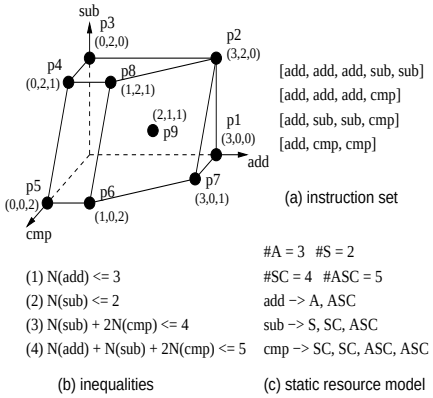


Figure 4.9: An instruction set having no SRM

tion $[add, add, sub, cmp]$ that is not part of the initial instruction set; it is depicted as point p9 in Figure 4.9 (a). As a result of the above theorem, we may conclude that the initial instruction set has no SRM. Point p9 is part of the plane through points p2, p6, p7 and p8. Thus, it is straightforward to verify that we cannot tighten the corresponding constraint expressed by inequality (4) in such a way that it excludes point p9 but none of the other points that do correspond to valid instructions. Although this is a negative result, it provides us useful information for instruction set design. If we could extend the initial instruction set with instruction $[add, add, sub, cmp]$, the resulting instruction set is guaranteed to have an SRM, which implies all the advantages discussed in this chapter.

Note that for certain important classes of instruction sets, it is always possible to derive an SRM using our method. One such class is the class of VLIW instruction sets, which serve as the interface for VLIW architectures that are used quite often in media processing. As a final remark, our method is a generalization of the methods presented in [23], [73], and [84].

4.6 Experimental results

In this section, we provide several experiments on constructing SRMs for instruction sets and applying SRMs to constraint analysis. The construction of the SRM and the application of the SRM to constraint analysis are automated. The first and second example show that our method can deal with complex instructions in DSPs and ASIPs. The third experiment proves that the SRM approach can be used to evaluate the restrictiveness of an instruction set. This provides quite use-

ful information for instruction set design. Loop folding can be applied efficiently with the SRM approach, which is exemplified by the fourth example. Finally, the SRMs for the reconfigurable instruction set architecture proposed in Chapter 3 can be obtained and tuned efficiently, which will yield a fast method for architecture exploration. All the experiments are performed on Pentium IV processor running at 1.6 GHz.

4.6.1 Complex instructions in DSPs and ASIPs

Often in an instruction set for a VLIW architecture, operation types are represented by a reservation table and are constrained by issue slots. In this case, approaches in [23] and [73] can translate the issue slot constraints to static resource constraints. However, the instruction sets of DSPs or ASIPs are designed with many irregularities. For instance, the basic instructions for the TMS320C25 are provided in Table 4.1. Complex instructions $[lt, pac]$, $[mpy, pac]$, $[lt, apac]$ are also introduced to exploit the limited parallelism in the data path. Such complex instruction constraints cannot be expressed as a reservation table, because not all the combinations of the operation types are valid, e.g., $[mpy, apac]$ is not a valid complex instruction. Our methods are not hampered by those irregularities.

Table 4.1: Basic instructions in the TMSC320C25 processor

instruction	implementation
LT	T register := memory value
MPY	P register := T register * memory value
PAC	ACCU := P register
APAC	ACCU := ACCU + P register
SACL	memory cell := ACCU

The TMS320C25 processor architecture is depicted in Figure 4.10 and all the instructions of TMS320C25 are shown in Figure 4.11 (a). This instruction set can be proved to have an SRM and it is shown in Figure 4.11 (b). We have implemented UTDSP benchmarks [74] on the TMS320C25 processor and the result is given in Table 4.2.

The second column shows the number of operations in each application. When complex instructions are not provided, no instruction-level parallelism (ILP) can be exploited. The scheduling result should be that the number of cycles equals the number of operations in each application. The "integer LP" column shows the number of instructions when ILP is exploited with complex instructions, which is

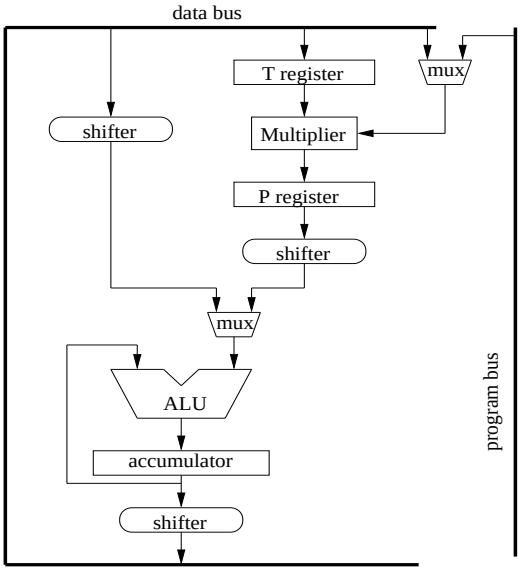


Figure 4.10: TMS320C25 architecture

[lt]	#SLM = 1, #SMP = 1, #SPA = 1 #SLMPA = 2
[mpy]	lt −> SLM, SLMPA
[pac]	mpy −> SLM, SMP, SLMPA
[apac]	pac −> SMP, SPA, SLMPA
[sacl]	apac −> SPA, SLMPA
[lt, pac]	sacl −> SLM, SMP, SPA, SLMPA
[lt, apac]	
[mpy, apac]	
(a) (complex) instructions	(b) SRM for this instruction set

Figure 4.11: TMS320C25 instructions and the corresponding SRM

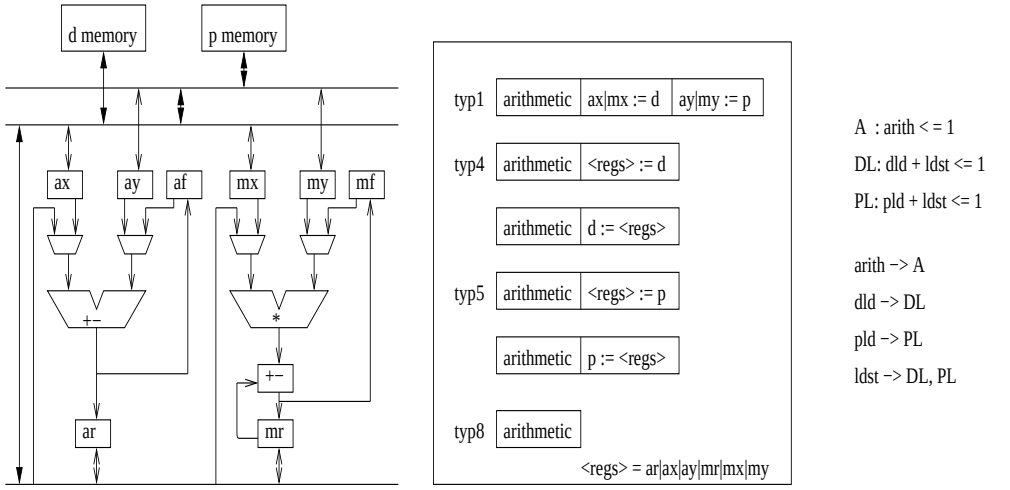
usually based on integer linear programming, an approach well-known to be time-consuming. For the benchmarks in Table 4.2 the execution time for scheduling with the "integer LP" method is in the order of seconds. Column "SRM" reports the number of instructions by constructing the SRM first and applying the SRM in our resource-constraint-based scheduler FACTS. Since the number of complex instructions is quite small, the runtime of constructing the SRM can be ignored. The time reported is the time required to obtain a schedule. As we can see in all examples, the SRM approach can generate very compact code equivalent to the optimal results in less than one second, which is very efficient. The SRM needs to be constructed only once for an architecture and can be used for any application. Moreover, the scheduling complexity is still linear with the number of nodes in the application, since the function between operation types and virtual resources is linear.

Table 4.2: Scheduling result of UTDSP benchmarks on TMS320C25 processor

benchmark	$\|V\|$	*integer LP [47]*	*SRM*	
		cycle	*cycle*	*t(ms)*
fft	24	20	20	920
fir filter	5	5	5	10
iir filter	17	14	14	350
lattice filter	14	11	11	60
lms fir filter	7	6	6	10

Another popular DSP is the Analog Devices' ADSP series. The Analog Devices' ADSP-21xx DSP contains parallel data paths of ALU, Multiplier / Accumulator and dual memory banks as depicted in Figure 4.12 (a). Because of the code size consideration, a highly encoded instruction format with maximal 24 bits is defined, which is depicted in Figure 4.12 (b).

In this encoding, parallelism in the data path is highly restricted. Since only one arithmetic operation is allowed in each instruction, we group the operations alu, mul and mac into one operation type $arith$. As we can see, two loads can be executed in parallel with one arithmetic operation, but they are limited to accessing dual memory banks D and P simultaneously and the loaded values can only reside in X and Y register group. We distinguish the dual memory loads from the general load/store operation type and they are denoted as dld and pld respectively. The other load/store operation type which can be executed in parallel with one arithmetic operation type is denoted as $ldst$ and it can access any of the registers in X, Y or R group. The generated SRM is illustrated in Figure 4.12 (c).



(a) ADSP−210x data path (b) ADSP−210x instruction format (c) corresponding SRM

Figure 4.12: Data path and instruction format of ADSP-21xx

Table 4.3: Scheduling of dspstone benchmarks on ADSP-21xx

benchmark	*bash [12]*		*ours*	
	cycle	*t(s)*	*cycle*	*t(s)*
complex multiply	6	1.11	6	0.01
complex update	9	1.76	9	0.03
iir filter	12	1.15	12	0.07
dot product	4	0.91	4	0.01
lattice filter	18	6.53	18	0.12

Using the corresponding SRM, we perform scheduling on a set of dspstone benchmarks and compare the results with the work from [12], which is shown in table 4.3. As we can see, we obtain the same scheduling results for all the benchmarks as in the work of [12] but with much less time. This is mainly because constraint analysis works faster than general schedulers. In addition, the SRM in this example is quite small after grouping of some operations into operation types. Mapping the operations in the DFG to the virtual resources will not increase the runtime too much. The advantage of this approach is that once the SRM is built, it is recognized as a set of resource constraints and all the constraints are permanently satisfied during scheduling; thus we solve the phase coupling problem.

4.6.2 Evaluating the restrictiveness of an instruction set

One of the merits of the SRM approach is that by performing instruction scheduling for applications with different SRMs, one can directly see from the results how restrictive an IS is. The experiments below perform the scheduling and register binding for benchmarks with the two instruction sets IS_1 and IS_2 in Section Figure 4.8. Assume that the hardware resources are ALU, MUL and $SHIFT$, each with two instances available. Each resource has a delay of one clock cycle. Table 4.4 lists the results for 'fdct', 'idct', 'ar' for AR filter and 'wdf' for fifth-order digital elliptical wave filter. The second and third column show the latency and register requirements in each benchmark (one register file for allocating all the values) assuming that all the given functional resources can be used freely in parallel. The fourth and fifth column give the result for IS_1 and the sixth and seventh column for IS_2. #in is the number of inequalities of the SRM, which corresponds to the number of (virtual) resources.

Table 4.4: Scheduling and register binding results for two instructions

	FS		IS_1		IS_2	
	L	RF	L_{SRM}	RF_{SRM}	L_{SRM}	RF_{SRM}
fdct	13	12	16	12	21	14
idct	14	11	15	11	20	12
ar	10	6	11	7	14	8
wdlf	16	7	16	7	19	11
#in	3		6		4	

From this table we can see that both the latency and register requirements, when the functional resources can be used freely, are minimum for all the examples. Although using the same resources, IS_2 is obviously more restrictive than IS_1, since all the scheduling results are increased quite substantially. Register requirements are also increased because when some operations are postponed during scheduling in order to meet the instruction set constraints, the values they consume have to be stored in registers for a longer time, which increases the register pressure.

4.6.3 Loop folding with the SRM approach

VLIW instruction sets have an SRM because of their orthogonal instruction format, such as the TMS320C62x architecture. The C62x has two identical data paths with four issue slots each. Each data path has 16 32-bit registers. Table 4.5

shows part of the instructions supported by TMS320C62x. L, S, D and M refer to four issue slots in one data path. Each issue slot contains one or more functional units. They are listed in the first column. In an integer adder functional unit, arithmetic operation add, sub, as well as comparison greater than $cmpgt$, comparison less than $cmplt$ can be executed. Data which are moved from one data path to another are supported by mov operation. Explanation for other operations can be found in [71].

Table 4.5: TMS320C62x instruction set

	L	*S*	*D*	*M*
Integer adder	add	add	add	
	sub	sub	sub	
	mov	mov	mov	
	cmpgt			
	cmplt			
Logic	and	and		
	or	or		
	not	not		
Shift		shl		
		shr		
Load/Store			ld	
			st	
Multiplier				mpy

Using the convex hull approach, virtual resources can be created for this instruction set. We will show that the SRMs can be applied to software pipelining easily, since they only provide static boundaries without modifying the scheduling and register binding algorithms themselves. Table 4.6 shows the scheduling and register binding results for dspstone benchmarks and table 4.7 shows these results for GSM channel and speech codec algorithms. In Table 4.7, 'convolution' and 'viterbi' are the two basic blocks in the channel codec. 'weight' and 'inverse' are two basic blocks for regular pulse excitation encoding, and'reflect' is the basic block for linear predictive coding in speech codec.

In Table 4.6 and Table 4.7, $|V|$ is the number of operations in each application. II is the minimal initiation interval calculated using our research tool FACTS with the SRM obtained for the TMS320C62x as input. The tool starts from an initial initiation interval yielding a feasible solution. By reducing the initial value and analyzing the feasibility iteratively, it finally obtains the minimum one. Column MII gives the initiation interval estimate using virtual resources, which is based

Table 4.6: Scheduling and register binding results for dspstone benchmarks

benchmark	$\|V\|$	*II*	*MII*	*RF*	*t(ms)*
complex update	16	9	8	5	20
convolution	5	2	2	4	10
dot product	8	8	5	3	10
matrix 1x3	27	12	12	7	330
real update	6	6	4	2	10

the similar method of resource-constrained minimum initiation interval estimate. This method is going to be explained in detail in Chapter 5 in this thesis. RF reports the register requirements under II. The last column gives the total runtime including scheduling and register binding. The results show that our tool FACTS can work with SRMs and obtain good results for software pipelining applied to industrial algorithms in acceptable run times. The runtime for "viterbi" is large compared to the others because FACTS is not optimized for handling large applications with many operations in the DFG using more than one virtual resource. These results indicate that the estimation method in Chapter 5 is quite accurate and could therefore be used for quick architectural exploration.

Table 4.7: Scheduling and register binding results for GSM speech algorithms

benchmark	$\|V\|$	*II*	*MII*	*RF*	*t(ms)*
invers	6	2	2	5	10
convol	14	5	5	8	20
reflect	19	7	7	6	170
weight	39	11	11	7	640
viterbi	55	19	19	10	15570

4.6.4 Reconfigurable instruction set processor architecture

In Chapter 3 we proposed a data path for a reconfigurable instruction set processor architecture with the corresponding instruction decoder. In this chapter, we can see immediately the advantage of this architecture since the SRMs for different instruction set configurations can be obtained efficiently through the approach discussed in this chapter. We illustrate the application of SRM to this architecture in the following example. Figure 4.13 originates from Figure 3.10. Some

modifications are applied. The number of functional units is reduced to half for reasons of convenience. The detailed connections from the instruction decoder to the functional units are shown. The type of a functional unit can be selected from the following set: ALU, MUL, SFT and $LDST$.

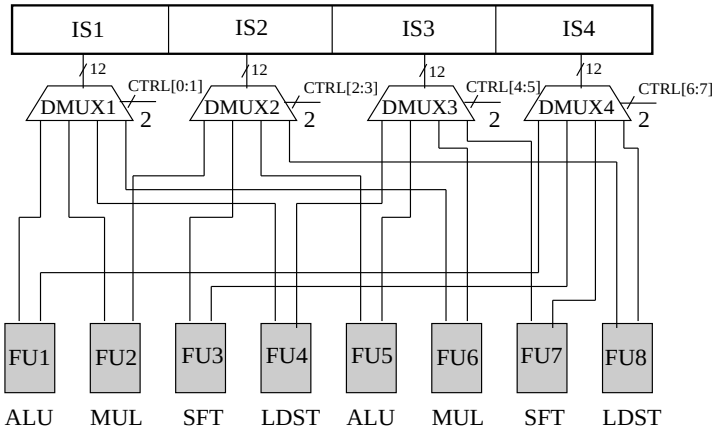


Figure 4.13: Proposed data path

Each demultiplexer is connected to a set of four functional units and at most 2 bits are needed to control the selection of a functional unit. In the case of full control, i.e. the demultiplexer selects all the functional units which are connected to it, we group all the functional units for each issue slot and represent them in a table, see Table 4.8.

Table 4.8: Issue slot table of reconfigurable instruction set with full control

$Slot_1$	$Slot_2$	$Slot_3$	$Slot_4$
ALU	MUL	LDST	ALU
MUL	SFT	ALU	SFT
LDST	ALU	MUL	SFT
MUL	LDST	SFT	LDST

In order to reduce the instruction width, the control of each demultiplexer does not fully cover the set of functional units. We assume two configurations: the demultiplexer selects the first two functional units of each set and the demultiplexer selects the second two functional units of each set. We call the two cases “upper half” and “lower half” respectively. For example, $DMUX1$ selects either $FU1$

or $FU2$ in "upper half" configuration. It selects either $FU4$ or $FU6$ in "lower half" configuration. The obtained SRMs for "full", "upper half" and "lower half" configurations are given in Table 4.9. In this table, $N(a)$, $N(m)$, $N(l)$, or $N(s)$ represent the number of addition/subtraction, multiplication, load/store or shift operations, which can be implemented on functional unit ALU, MUL, $LDST$ or SFT respectively.

Table 4.9: SRMs of full and half control reconfigurable instruction sets

$configuration$	SRM
full	N(a) $\leq$ 2 N(m) $\leq$ 2 N(l) $\leq$ 2 N(s) $\leq$ 2 N(a) + N(m) + N(l) + N(s) $\leq$ 4
upper half	N(l) $\leq$ 1 N(s) $\leq$ 2 N(m) $\leq$ 1 N(a) + N(l) $\leq$ 2 N(a) + N(l) + N(s) $\leq$ 3
lower half	N(a) $\leq$ 2 N(l) $\leq$ 1 N(s) $\leq$ 1 N(a) + N(m) + N(l) + N(s) $\leq$ 4

The scheduling results of the three configurations on several benchmarks are shown in Table 4.10.

From this table, we can conclude that in general fully controlled reconfigurable instruction set results in a smaller number of "extra virtual resources" (virtual resources exclude functional resources), which is time efficient for scheduling and register binding. Although it results in fast schedules for all the benchmarks, it needs more bits to encode for each instruction, thus results in somewhat larger code size. On the other hand, a half controlled reconfigurable instruction set produces more extra virtual resources than full controlled configuration, which will result in possibly longer schedule length and cost more run time for scheduling and register binding. Since code size depends on both the instruction width and code length, we cannot give direct conclusion on the impact on the code size, although we can already see that half of the examples have the same schedule lengths for the three configurations. We estimate the code size based on the following assumption of the instruction format: assuming that in each issue slot there

Table 4.10: Schedule length(cyc), code size(cs) and run time(t) for benchmarks on "full", "upper half" and "lower half" configurations

benchmark	*full*			*upper half*			*lower half*		
	cyc	cs	t(ms)	cyc	cs	t(ms)	cyc	cs	t(ms)
complex multiply	6	384	30	6	360	20	6	360	20
complex update	7	448	40	7	420	70	7	420	50
dot product	5	320	10	5	300	10	5	300	10
matrix	8	512	230	9	540	230	9	540	250
fft	10	640	70	10	600	100	10	600	60
iir	5	320	30	5	300	40	5	300	30
invers	7	448	0	7	420	0	7	420	0
convolu-tion	5	320	10	7	420	50	5	300	40
reflect	13	832	80	13	780	120	13	780	80
weight filter	18	1152	540	18	1080	790	18	1080	550
viterbi	12	768	680	16	960	1250	12	720	720

are three operand fields, two source operands and one destination operand. Each operand field is encoded with 3 bits. The opcode field is encoded with 5 bits. Depending on the configuration, one or two bits is used for the selection of the demultiplexer. From the reported code size, we can see that in most of the examples code size is reduced in "upper half" and "lower half" configurations. It seems that in this architecture the "lower half" configuration is slightly better than the "upper half" configuration. Therefore we conclude that by carefully configuring the data path and the instruction decoder, we can possibly gain both in performance and in code size of a set of applications. This architecture template provides an efficient solution to balance between the code size and processor performance. The SRM can be efficiently used to assess the of control bits and connection on code size and execution speed.

4.7 Conclusions and discussions

In this chapter, we provide the method of static resource models for modeling instruction set constraints. The phase coupling problem of instruction selection and other code generation phases is relieved by modeling the instruction set constraints and integrate them with the resource constraints. This allows an efficient compilation for certain irregularities in the instruction set and the data path, which is important for retargetability. The main advantages of the SRM approach are the following:

- Run time. The SRM of a processor's instruction set architecture has to be computed only once. On the other hand, instruction selection has to be performed for each basic block of the application. Consistency checking [39] has to be performed for each operation in the basic block.
- Schedule freedom. The scheduler is not restricted by a specific instruction selection phase and therefore has more opportunity to minimize the register requirements.
- Handling of pipelined schedules. The scheduler also has the opportunity to produce loop-pipelined schedules (also called loop folding or software pipelining). In the DSP world this is a must to exploit the instruction-level parallelism.
- Any resource-constrained scheduler can be used without much adaptation.

Furthermore, the SRM approach can be adapted for reconfigurable data path and instruction set design easily since the design details are transferred to a set of constraints which are visible and easily tunable by the designer.

The disadvantage is that the approach might be time-consuming for very irregular architectures since those architectures will yield large sets of virtual resource constraints.

Chapter 5

Instruction Set Design with the SRM Approach

5.1 Introduction

ASIPs offer the possibility to exploit the characteristics of the application (-domain) and thus gain considerable savings in silicon area, power consumption, and code size. There are roughly three ways to tune an ASIP core to an application:

- By synthesizing an infrastructure of communication (busses) and storage (registers) which is just sufficient for the application.
- By hardware acceleration [70]. The data path is upscaled by functional units that perform course grain functions typical for the application. An example is a butterfly unit in an FFT processor.
- By minimizing the width of the instructions required to control a given data path [83]. One way is to encode frequently occurring (sequences of) operations with short instruction words. Another possibility is to limit the number of instructions by restricting the combinations of operations that the data path can execute in parallel.

These ways can also be combined. Hardware acceleration potentially offers the largest benefits on all accounts, especially for applications that contain much regularity. It is also the most complex method for the designer because it requires changes in the communication and storage hardware and the design of the dedicated functional units every time a new application is embraced. The above mentioned ways all have the same severe drawback: They imply the necessity to "recognize" in the application those instructions to be supported in the tuned instruction set.

As we mentioned in the previous chapter, the SRM approach can be twisted to obtain "minimal" instruction sets by restricting the set of combinations of operations that the data path can execute in parallel. Instruction sets can be modeled in terms of *virtual* resources, easily interpreted by classic schedulers accounting for resource constraints. This yields an alternative machine model with virtual resources, which allows efficient resource constrained compilation with well understood and widely available compilation tools, rather than the poorly performing compilers based on instruction selection.

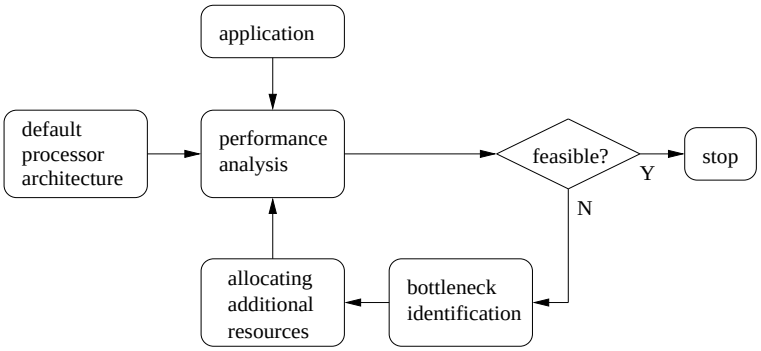


Figure 5.1: Design flow of ASICs

The SRM approach also enables instruction set design (-space exploration) with an equally well-understood and proved method used in the High-Level Synthesis (HLS) of ASICs [8] for a long time. This method, illustrated in Figure 5.1, analyzes the time critical loops for shortages of processor resources required to obtain the target schedule throughput. These shortages can be identified by scheduling the loop and examining the *load diagrams* of the functional resources. The load on critical resources is then relieved by allocating additional resources. The potential use of this method for instruction set design is based on the observation that both real functional resources **and** virtual resources can be allocated when considering the SRM model of an instruction set. The addition of virtual resources results in an extension of the instruction set. We presume therefore that the SRM view on an instruction set allows instruction set design in terms of allocating resources just sufficient to efficiently execute the critical loops.

This chapter discusses the instruction set design problem by applying the SRM approach proposed in previous chapter. Section 5.2 presents the problem definition and the approach. Section 5.3 discusses performance analysis and bottleneck identification. Modification of the identified SRM bottleneck is discussed in Section 5.4 and a case study is given in Section 5.5.

5.2 Problem definition and approach

Usually the instruction set design process is performed independently from the compiler. Thus it could happen that although a good processor architecture is generated, it can not produce the desired performance for applications even if a lot of effort is put on generating an efficient compiler. It is a challenge to design an instruction set for an ASIP that can be encoded using a restricted number of instruction bits, while still offering a sufficient degree of parallelism for critical functions in the target application. We consider the following instruction set design problem.

Problem Definition 5.1 *Given a set of time critical loop kernels with the corresponding throughput constraints and a target instruction width for the $ASIP$, design an instruction set and the corresponding SRM such that the throughput constraints can be satisfied.*

Noticing the similarity between the functional resources and the virtual resources, we propose an optimization flow similar to the flow in Figure 5.1 for allocating functional resources in high-level synthesis.

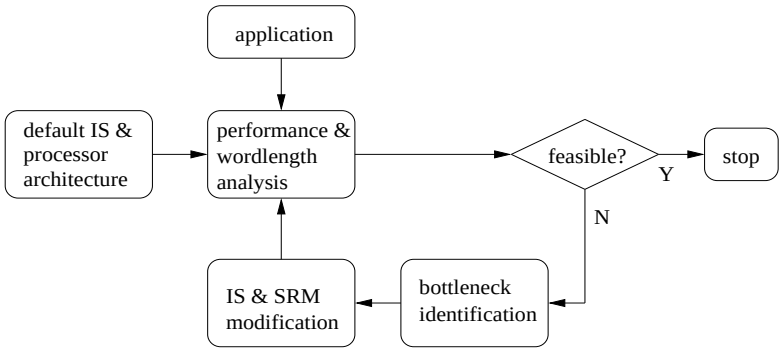


Figure 5.2: Design flow of instruction set for ASIPs

In this optimization flow, we start with a *default instruction set* and *processor architecture*. Subsequently, the performance on the critical loops is analyzed, which is explained in more detail in the next section. If the performance is insufficient, we look for the responsible (virtual) resources in a step called *bottleneck identification*. The SRM is subsequently modified by allocating additional instances of these virtual resources. The essential difference between Figure 5.2 and the method of the high-level synthesis flow in Figure 5.1 is that we also consider

the instruction width as a criterion in the design process to evaluate the modifications applied to the SRM.

5.3 Performance and bottleneck analysis

Similar to the high-level synthesis approach, performance analysis can be done either fast or accurate. An accurate analysis is obtained by actually scheduling the critical loops and examining the load diagrams. A load diagram is a diagram to record the resource usages with respect to value lifetimes and program counter. These load diagrams enable the designer to identify critical resources. A load diagram example is shown in Figure 5.3. In this figure, the bottom x-axis corresponds to the program counter, the top x-axis corresponds to the potential. Potential is especially useful for loops. Operations in different loop iterations can be allocated in the same time slot and this time slot is annotated with potential. The column axis shows the available resources. Potential with negative number is the initiation section. Potential between dashed line and dotted line is loop prologue. Potential within dotted lines is loop kernel and potential after dotted line is epilogue. This figure shows that quite often the resources are heavily used in loop kernels. Alternatively, the performance of the critical loops can be estimated in a fast way by considering a well-known lower bound based on available (virtual) processor resources, which is explained next.

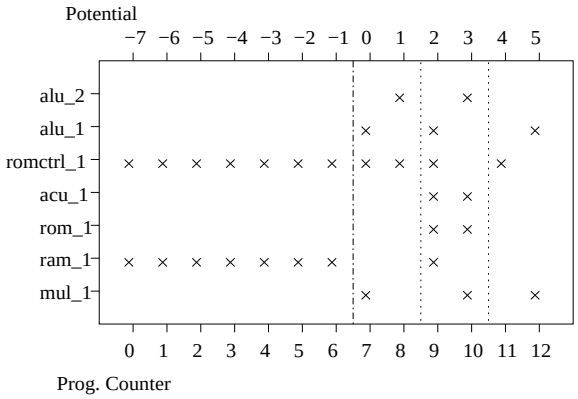


Figure 5.3: A load diagram

When applying software pipelining techniques to loop kernels, the *initiation interval (II)* is an important criterion for measuring the performance. An II is the period between the start times of the execution of two successive loop-body

iterations. The *minimum initiation interval (MII)* is the lower bound of the II. The MII can be determined either by a critical resource that is fully utilized, i.e. the *resource-constrained MII (ResMII)*, or a critical chain of dependencies running through the loop iterations, i.e. *recurrence-constrained MII (RecMII)*. Since we assume that recurrence DFGs are all transformed into non-recurrence DFGs in our application, we only focus the first one. The ResMII is derived by calculating, in total, the usage requirements for each resource imposed by one iteration of the loop.

Suppose a loop containing 14 add operations is mapped on a data path containing three adders and each adder takes one clock cycle to execute. Then we need at least $\lceil \frac{14}{3} \rceil = 5$ clock cycles to execute the loop iteratively. By executing this calculation for every available resource and finding the maximum one, we obtain the lower bound ResMII on the initiation interval II of a pipelined schedule of the loop. The general experience is that this bound is very tight [60]. The lower bound indicates the critical functional resource in the data path. This lower bound estimate can therefore be used for bottleneck identification.

In case of instruction set constraints and the corresponding SRM, virtual resources must be taken into account for the performance estimation. The ResMII estimation is no longer obtained by simply totaling the resource usages for each resource because some operations are mapped to more than one instance of a virtual resource. Thus the ResMII estimation has to be modified to account for the multiple usage of virtual resources [85]. Suppose a virtual resource r_j with $\#r_j$ instances available, and r_j is used by a set of operation types $\{op_i, i = 1, \cdots, n\}$. Also assuming that in the DFG the number of operations using operation type op_i is n_i, then ResMII is computed as:

$$\mathrm{ResMII}_{\mathrm{SRM}} = \underset{r_j \in R}{\mathrm{MAX}}\left(\frac{\sum_{i=1}^{n} a_{i,j} n_i}{\#r_j}\right) \tag{5.1}$$

where R is the set of virtual resources and $a_{i,j}$ is the i-th element in the j-th row corresponding to virtual resource r_j in the matrix A in equation 4.3 and $\#r_j$ is actually the jth element in vector $\vec{b}$.

For example, for the instruction set in Figure 4.8 (b), inequality (6') in Figure 4.8 (d) indicates that the $shift$ operation uses two instances of the resource MS, of which four are available. For a DFG in Figure 5.4, the number of operations n_{add}, n_{mul}, n_{shift} using operation types add, mul and $shift$ are 2, 6 and 4 respectively. Assume functional resources $adder$, $multiplier$ and $shifter$ are available, each with two instances. Thus the estimated lower bound for the initiation interval with respect to the functional resource constraints can be calculated as follows.

$$\begin{aligned} \mathrm{ResMII_{FU}} &= max(\lceil\frac{n_{add}}{\#adder}\rceil, \lceil\frac{n_{mul}}{\#multiplier}\rceil, \lceil\frac{n_{shift}}{\#shifter}\rceil) \\ &= max(\lceil\frac{2}{2}\rceil, \lceil\frac{6}{2}\rceil, \lceil\frac{4}{2}\rceil) \\ &= 3 \end{aligned} \tag{5.2}$$

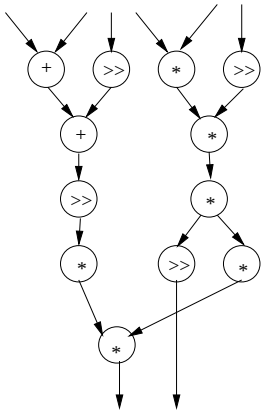


Figure 5.4: An example DFG for MII estimation

While the DFG in Figure 5.4 is executed by instruction set IS_2, the minimum initiation interval should be modified by the instruction set constraints, i.e. the constraints from virtual resources A, M, AM, MS, AMS obtained from 4.8 (f). The modified $\mathrm{ResMII_{SRM}}$ is shown in 5.3 in details.

$$\begin{aligned} \mathrm{ResMII_{SRM}} &= max(\lceil\frac{n_{add}}{\#A}\rceil, \lceil\frac{n_{mul}}{\#M}\rceil, \lceil\frac{n_{add}+n_{mul}}{\#AM}\rceil, \lceil\frac{n_{mul}+2*n_{shift}}{\#MS}\rceil, \\ &\quad \lceil\frac{n_{add}+n_{mul}+n_{shift}}{\#AMS}\rceil) \\ &= max(\lceil\frac{2}{2}\rceil, \lceil\frac{6}{2}\rceil, \lceil\frac{2+6}{3}\rceil, \lceil\frac{6+2*4}{4}\rceil, \lceil\frac{2+6+4}{4}\rceil) \\ &= 4 \end{aligned} \tag{5.3}$$

This estimation yields the new value of the initiation interval which is one clock cycle longer than the original one based on the functional resources. This lower bound is very tight; we can apply the loop folding technique directly by first mapping the operations to the virtual resources in Figure 4.8 (f). Figure 5.5 shows the optimal scheduling result and the virtual resource usages in the loop kernel.

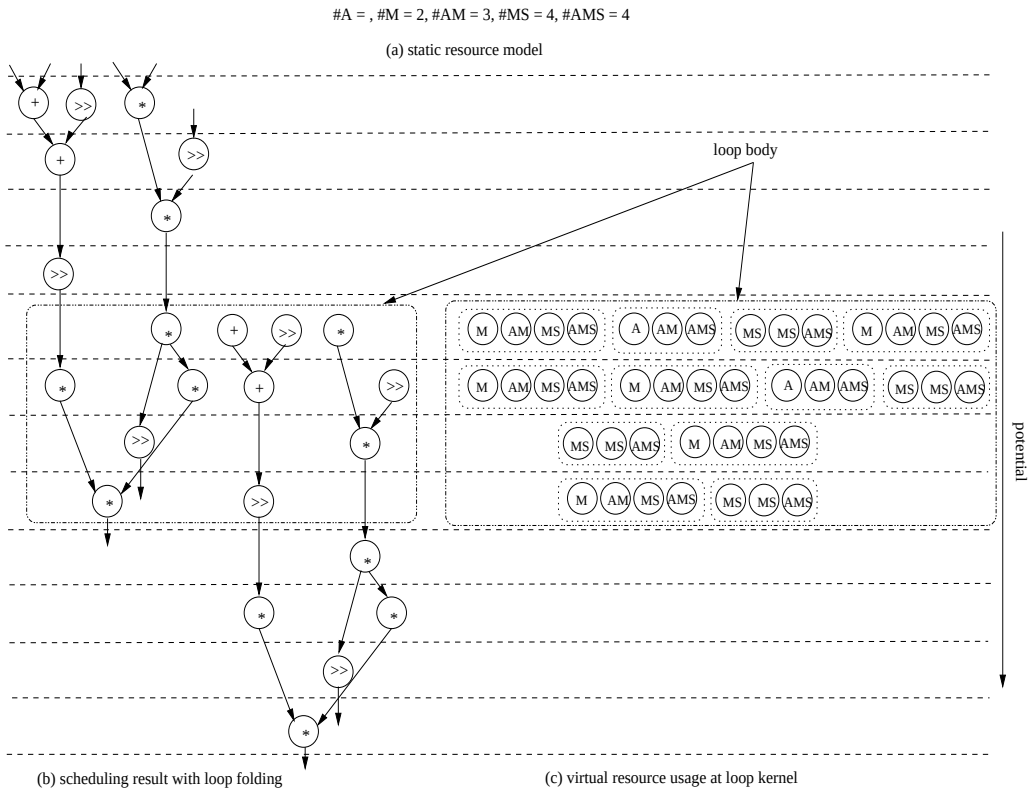


Figure 5.5: Scheduling result for a loop kernel and the virtual resource usage

Like in the bottleneck analysis of high-level synthesis of ASICs in Figure 5.1, the bottleneck of performance now can be identified from the updated lower bound estimation and can be relieved by allocating additional (virtual) resources. In addition to the allocation of additional resources, in our instruction set design flow we also have the possibility to decrease the resource usage of a critical resource in order to relieve the bottleneck. Therefore it is more convenient to consider the inequalities, because they describe both the resource availability and for each operation type, and the usage of that resource. The way that we relieve the bottleneck is explained in the following section.

5.4 Modification of the SRM

Consider again the example of Figure 5.4. In the previous section, virtual resource MS has been identified as the bottleneck since it is the largest value in the ResMII estimation. Recall that each coefficient on the right hand side of an inequality is the number of instances of a virtual resource. Each left hand side coefficient of an operation type reflects the usage of the corresponding virtual resource. Thus the larger the coefficient, the more virtual resources will be used by the considered operation type. We take into account two possibilities to modify the SRM given in Figure 4.8 (f). Consider inequality $(6')$ in Figure 4.8 (d).

- One possibility is to increase the number of instances (i.e., the right hand side of the inequality) of virtual resource MS. This change means that possibly more parallel operations are allowed to be exploited under the relaxed constraints; it might result in an increase of the total instruction width.

- A second possibility is to decrease the largest use (the $shift$ operation) of the resource MS. We consider the largest use because it has the largest impact on the ResMII lower bound. Also this change might result in an increase of the total instruction width; it allows a better balancing of the different operations within the possibly new instruction width constraints because it tends to equalize the weights in the left hand side of an inequality.

These two possibilities to adapt an instruction set by modifying its SRM are generally applicable. The idea is always to first identify the critical virtual resources causing a performance bottleneck and then to relieve the bottleneck through the proposed techniques. In case there is more than one bottleneck, we provide the following heuristics for the bottleneck selection.

- Select the virtual resource implying a smaller initiation interval after applying the modification mentioned above since it effectively improves the performance.

- In case the bottlenecks result in the same performance after the modification, select the virtual resource causing a smaller instruction width increase.

Since modifying the left hand side coefficients of an inequality does not necessarily increase the instruction width, we prefer performing this modification strategy first. Furthermore, since the larger the coefficients, the more virtual resources an operation type will use, we prefer starting with a larger coefficient and hope to obtain a more balanced resource usage with regard to other resources after modification.

An example in Table 5.1 illustrates the design process. In this example we consider a loop with 9 mul operations and 6 add operations. The initial instruction set and the corresponding inequalities are shown in Figure 4.7. Notice that the operation type add and mul are encoded with 8 bits and 10 bits respectively, and the total instruction width is restricted to 28 bits. The performance requirement of the loop under consideration is given as $II = 5$.

Table 5.1: Modification of the SRM

		initial design	*right-side adjust*	*left-side adjust*	
				second coef	*fi rst coef*
inequality		2N(add)+3N(mul) ≤ 6	2N(add)+3N(mul) ≤ 8	2N(add)+2N(mul) ≤ 6	0.5N(add)+3N(mul) ≤ 6
SRM	A	3	3	3	3
	M	2	2	2	2
	AM	6	8	6	6
resource usage	add	A, 2 AM	A, 2 AM	A, 2 AM	A, 0.5 AM
	mul	M, 3 AM	M, 3 AM	M, 2 AM	A, 3 AM
IS		[add, add, add] [add, mul] [mul, mul]	[add, add, add] [add, add, mul] [add, mul, mul]	[add, add, add] [add, add, mul] [add, mul, mul]	[add, add, add, mul] [mul, mul]
MII		7	5	5	5
wordlength		24 bits	28 bits	28 bits	34 bits

In Table 5.1, the second column corresponds to the initial design depicted in Figure 4.7 (c). The virtual resource AM associated with the third inequality is identified as the bottleneck, which bounds II to 7. In the third column, we evaluate the decision to modify the right hand side of the bottleneck by increasing the instances to 8. As a result of this modification, II is now bounded to 5. New instructions $[add, add, mul]$ and $[add, mul, mul]$ are added to the instruction set, thereby increasing the instruction width to 28 bits. In the fourth and fifth columns we evaluate the decision to modify the left hand side coefficients. The fourth column modifies the bottleneck by decreasing the larger coefficient. The lower bound on the II is now 5 and the instruction width increases to 28 bits. The fifth column reduces the smaller coefficient. It also meets the performance requirements, while the code size in increased to 34 bits. From this example, we see that although

the inequalities are different, the design in the third and fourth column both meet the performance requirements and code size reqquirements. We consider a more elaborate example in the next section.

5.5 Case study

In this section, we demonstrate the practical applicability of our instruction set design flow.

[add, mul, mul]
[add, sub, sub, mul]
[add, add, sub, load]
[add, add, sub, mul]
[add, add, add, mul]
[add, add, add, sub, sub]

Figure 5.6: An example instruction set

Table 5.2: The corresponding SRM of the instruction set in Figure 5.6

	inequality	*SRM*	#	*MII*
1	N(a) + N(l) $\leq$ 3	AL	3	5
2	N(s) + N(l) $\leq$ 2	SL	2	5
3	N(m) + 2N(l) $\leq$ 2	ML	2	8
4	N(s) + 2N(m) + 3N(l) $\leq$ 4	SML	4	8
5	N(a) + 2N(m) + 3N(l) $\leq$ 5	AML	5	7
6	N(a) + N(s) + 2N(m) + 2N(l) $\leq$ 5	ASML	5	7

For the instructions given in Figure 5.6 the corresponding SRM in Table 5.2 can be obtained using the approach in Section 5.3. We use the fast method for performance evaluation explained in Section 5.3 rather than performing detailed scheduling. Since the topology of the DFG of the loop is irrelevant for this analysis, we list only the number of operations and their resource usages. We assume the loop contains 9 add operations, 3 sub operations, 4 mul operations and 6 $load$ operations; add and sub are encoded with 8 bits each, mul and $load$ with 16 bits. For reasons of convenience, we abbreviate add, sub, mul and $load$ as a, s, m and l. The instruction width is given to be constrained to 40 bits. The fourth column

gives the number of instances of each virtual resource and the fifth column lists the estimated initiation interval contribution according to equation 5.1 for each virtual resource. Assuming the required overall II is 6, this design is far below the performance requirements.

Table 5.3: Modification for candidates (3) and (4)

	new inequality	*MII*	*extra instructions*	*wordlength*
3L4L	N(m) + N(l) $\leq$ 2	5	[m,l]	40
	N(s) + 2N(m) + 2N(l) $\leq$ 4	6		
3R4R	N(m) + 2N(l) $\leq$ 3	6	[m,l]	40
	N(s) + 2N(m) + 3N(l) $\leq$ 5	6	[s,m,m]	
3L4R	N(m) + N(l) $\leq$ 2	5	[m,l]	40
	N(s) + 2N(m) + 3N(l) $\leq$ 5	6	[s,m,m]	
3R4L	N(m) + 2N(l) $\leq$ 3	6	[m,l]	40
	N(s) + 2N(m) + 2N(l) $\leq$ 4	6		

Table 5.3 shows the different results by applying the modification methods in Section 5.3 to the bottleneck candidates (3) and (4). The first column refers to the design decision under evaluation. For example, '3L4R' represents the decision to modify the left hand side of inequality (3) and the right hand side of inequality (4). The second column presents the modified inequalities. The third column estimates the II according to the new SRM. Because of the modification, the resource constraints are relieved, and subsequently more instructions are allowed. The fourth column gives the new instructions besides those already provided in Figure 5.6. The fifth column calculates the wordlength with the new instruction set.

From Table 5.3 we can see that all designs sufficiently reduce the bottleneck. We choose the design of row three for next iteration for modifying the virtual resources because it gives the best combination of performance improvement and extra instructions. The next identified bottlenecks are virtual resources (5) and (6) in Table 5.2 because their II- estimate is 7, which is still greater than the required. The same procedure is repeated and shown in Table 5.4. From this figure, we can see that the second and third design exceed the wordlength limitation and have to be omitted. The first and fourth design meet both the timing and code size constraints and are acceptable.

This example shows that the SRM approach provides a good basis for tuning performance and code size for the purpose of instruction set design. This administrative approach is quite efficient for current embedded processor design because without looking at the detailed encoding and without performing exact scheduling for the applications the designer can already assess which adjustment should be

Table 5.4: Modification for candidates (5) and (6)

	inequality	*MII*	*extra instructions*	*wordlength*
5L6L	N(a) + 2N(m) + 2N(l) $\leq$ 5	6	[s,m,m]	40
	N(a) + N(s) + 2N(m) + N(l) $\leq$ 5	6	[a,m,l]	
5R6R			[a,m,l]	
	N(a) + 2N(m) + 3N(l) $\leq$ 6	6	[a,s,m,m]	48
	N(a) + N(s) + 2N(m) + 2N(l) $\leq$ 6	6	[a,a,m,m]	
			[a,a,s,s,m]	
			[a,a,a,s,m]	
5L6R			[a,m,l]	
	N(a) + 2N(m) + 2N(l) $\leq$ 5	6	[a,s,m,m]	48
	N(a) + N(s) + 2N(m) + 2N(l) $\leq$ 6	6	[a,a,s,s,m]]	
			[a,a,a,s,m]	
5R6L	N(a) + 2N(m) + 3N(l) $\leq$ 6	6	[s,m,m]	40
	N(a) + N(s) + 2N(m) + N(l) $\leq$ 6	6	[a,m,l]	

made.

5.6 Conclusions and Discussions

In this chapter, we propose a methodology for designing the instruction sets for ASIPs to meet the performance requirements as well as the code size constraints. This is performed iteratively by tunning the instruction set constraints which are represented as inequalities obtained from the SRM approach. For applying this method efficiently, we perform a fast performance estimation for loop kernels by calculating the minimum initiation interval based on virtual resources. From the estimation the bottlenecks to performance are identified and are relieved either by increasing the amount of virtual resources, or by reducing the virtual resource usage of operation types. Subsequently, new instructions are obtained and code size is evaluated for the new instruction set. Several heuristics are given for how to tune the instruction set constraints represented as inequalities. Further discussion and improvement includes:

- Tuning the instruction set using the two heuristics in Section 5.4 might result in different inequalities but the same new instruction set. This is understandable since instructions are integer points in the operation type space.

- In Section 5.4, for most of the implementation, the modification of the instruction set constraints and coefficients are all based on integer numbers

and the step for tuning is set to one. This is not necessarily a constraint, although it remains to see in practice how small a step can be for an efficient design for large applications.

- The approach always takes the larger coefficient on the left hand of an inequality for modification. This is based on the performance improvement criteria. Performance estimation is always performed before code size evaluation. This is because our main goal is to meet the performance requirements. It is possible to tuning all the coefficients at the same time, although it will be very difficult to predicate the influence on the performance from all the modifications.

- Since there are multiple points (new instructions) in the space that might be included in the new instruction set, it is more efficient to analyze those points first and include the best one for the balance of performance and code size, and then tuning the inequality. Although this is more efficient for small examples, for larger instruction set with many operation types, the task to analyze which point is a good candidate for balancing the performance and code size is very complicated.

- In this chapter, we calculate the code size constraints based on the worst-case code size for all the instructions. This is reasonable for small ASIPs and DSPs which usually contain several functional units, since those functional units are almost always been fully exploited by the instructions. For larger processors, such as VLIW type, this is very inaccurate, since NOPs appear quite often in the instructions. In fact many code compression techniques are developed for those NOPs to reduce the code size.

Chapter 6

Limited Address Range Architecture

6.1 Introduction

Conventional general purpose VLIW architectures usually exhibit one central register file as the storage unit. While this is convenient for compilation, the use of one central register file contributes to a large operand encoding field, subsequently large code size and high power consumption in the ROM/RAM which stores the application code. In addition, a large register file typically increases the number of register file read/write ports needed and leads to complex wiring, which results in high power consumption. The increases in wiring also affect cycle time and cause long execution delay. New designs usually partition the architecture into clusters. Each cluster contains several functional units and a local register file [15] [41]. The code size is reduced with a large amount. However, the communication of values among different clusters has to be supported by extra hardware, including buses, and probably separate copy operations have to be inserted [28] [45]. The latter complicates the design of a compiler and is not coherent with the small code size purpose.

Code size in DSPs and ASIPs is small compared to general purpose VLIW architectures. This is obtained by introducing irregular data paths and connections, as well as highly encoded instruction sets. Not only the number of parallel operations can be restricted by the encoding, but also the available number of registers for storing a value can be constrained to a small number. Often each read or write port of a functional unit is connected to a small register file with only a few registers, and it is the task of compilers to move data among those different register files.

In this chapter, we generalize the idea of limiting the amount of registers to

be accessed by a functional unit to certain ranges and propose the so-called limited address range (LAR) architecture [87]. Since encoding operands is costly in instructions, which is directly related to the number of registers, we restrict the encoding range to a subset in a register file. Instead of treating the subsets as independent register files, we allow overlap among different ranges. Communication between functional units can be put into the commonly addressable registers. This will reduce the communication cost significantly. However, it will introduce a new phase called address range assignment. In order to overcome the phase coupling between address range assignment and other tasks in code generation, the address range constraints are integrated with timing, resource and register file constraints. Efficient search space pruning techniques are used to prevent decisions that inevitably lead to violations with those constraints.

Figure 6.1 shows an example of the LAR architecture. In this figure, register files RF has 14 registers and is grouped into two sets S_1 and S_2, each contains 8 registers. Functional units FU_1 and FU_2 read and write values within range S_1, and functional units FU_3 and FU_4 read and write values within range S_2. Ranges S_1 and S_2 share registers $r1$ and $r2$. Values produced by FU_1 or FU_2 and consumed by FU_3 or FU_4 can be stored in $r1$ or $r2$, and vice versa. No extra hardware is needed and no extra move operations have to be inserted unless the number of overlapping registers is not large enough. Assume that an opcode is encoded with 5 bits. An instruction with three operands will cost 17 bits for the central register file architecture. Alternatively, it will cost 14 bits for the LAR architecture. Thus the saving of the encoding is 17.65%.

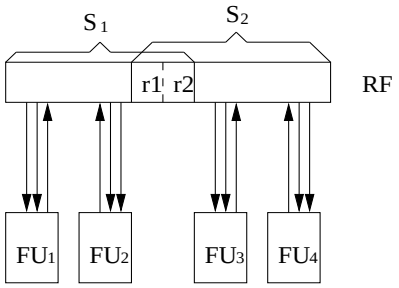


Figure 6.1: Limited address range architecture

In Chapter 4 we have developed the SRM approach for modeling the instruction set constraints and combine them with resource and timing constraints in an application. In this chapter, we extend this concept for the LAR architecture. Each range is viewed as a virtual register file and the number of registers in this range is identical to the virtual resource constraint. All those virtual register file

constraints are integrated into a uniform conflict graph and conventional coloring algorithm can be used for register allocation.

The outline of this chapter is as follows. Section 6.2 discusses the related work. Section 6.3 presents the problem statement and approach. Section 6.4 discusses the conflict graph construction and annotated conflict graph analysis. Experimental results are given in Section 6.6 and code size and performance are compared for different architectures.

6.2 Related work

Although a lot of work has been done on reducing the code size in VLIW architectures by partitioning the data path into clusters with local registers, little work has been done on using global registers for the communication among clusters. MAJC, a scalable microprocessor from SUN [50], is one of them. In MAJC, multiple processors reside on a single chip. Each processor contains many processor units and each processor unit usually contains four functional units. Each functional unit is self-contained and has a local register file, local wiring, local control (e.g. instruction decode logic) and state information. Functional units share global registers in a processor unit. Local registers which are specific to a functional unit are not accessible to other functional units. Register file size is variable and implementation specific. It can vary from 32 to 512. Assuming there are 64 global registers. Assume also there are 64 local registers for each functional unit. The total number of registers is $64 + 4 \times 64 = 320$, while the number of registers available for a functional unit is $64 + 64 = 128$. Thus the addressing cost for each operand field is reduced from 9 bits to 7 bits.

In addition to the encoding reduction, the number of read/write ports is greatly reduced. This is depicted in Figure 6.2 (b). In Figure 6.2 (a) a general VLIW architecture with a central register file is depicted. For the central register file, consider three read operands and one write operand (3R+1W) are involved for encoding each functional unit, totally 16 read/write ports are needed for the register file. The MAJC architecture is logically represented in Figure 6.2 (b). For each functional unit, besides accessing its local register file with three read ports and one write port, it can also be accessed by the other three functional units through a global register file. Therefore, seven ports, i.e. three read and four write ports (three from other functional units) for each register file is required and the number is greatly reduced compared to 16 ports for the central register file. Since the area of a register file is proportional to the square of the number of ports, this results in a smaller total register file area. It also enables easier register file fabrication, and reduces cycle time delays.

The MAJC architecture does not require much modification in the conven-

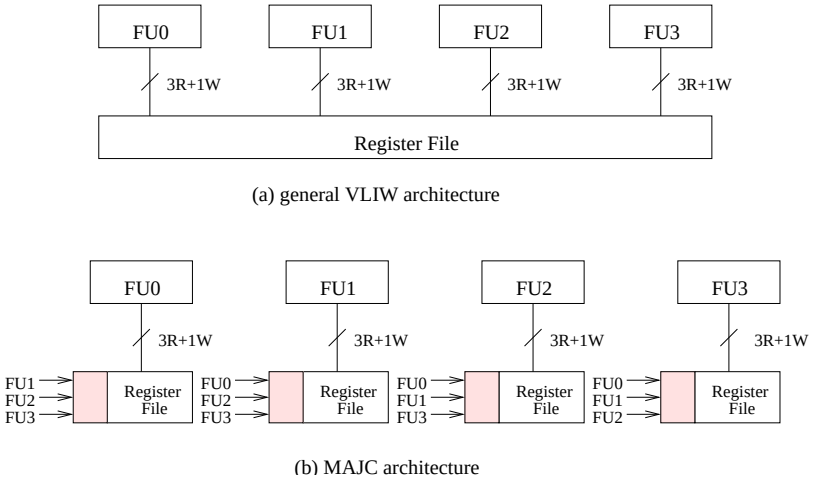


Figure 6.2: General VLIW architecture and MAJC architecture

tional compiler, since the number of registers in the global register file is normally enough for the communication among functional units. In addition, the functional units and general purpose register files in the MAJC architecture are data type agnostic. This provides more registers for applications that involve dedicated data type processing and significantly improves the performance. Further, it provides the compiler with the flexibility to allocate any type of data to any register.

In Cydra 5 [62], the context register matrix, which is a matrix of certain amount of registers, is provided to dynamically allocate iterations of loops at runtime. Each iteration is allocated in an iteration frame. Since the number of registers in the context register matrix is finite, the iteration frames for past iterations must be deallocated at the same rate as the new frames. This is necessary for loop variants. However, for loop invariants, which are only used but never computed, this will cause them to be overwritten unless they are copied in each iteration. To avoid those copies, General Purpose Register file (GPR) is provided with global registers to all the iterations. In a single cycle, it can be read by any number of functional unit input ports, but can be written only by one output port of the functional units.

6.3 Problem statement and approach

In this section we define the scheduling and register binding problem for the LAR architecture. We decompose the problem and construct a block diagram of the

global approach. Our problem statement is as follows.

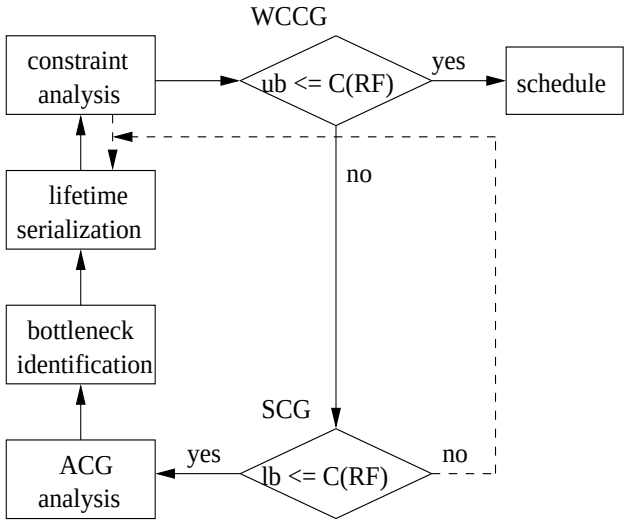


Figure 6.3: Global approach for LAR architecture

Problem Definition 6.1 *Given a DFG, resource constraints, a latency L, an initiation interval II, a register file RF with its capacity $C(RF)$, this register file can be subdivided into address ranges and functional units can access different ranges of the register file, find an assignment of values to registers and a schedule such that all the timing constraints L and II, resource constraints and address range constraints are satisfied.*

The global approach is based on the work in [4] with some additions and is depicted in Figure 6.3. As in [4], it is decomposed into several steps since decisions affect the search space in both the scheduling domain and the register allocation domain. Here an additional step called *ACG analysis* is inserted and will be explained later in Section 6.5.3. The central part, the *constraint analysis*, generates additional precedence constraints that are implied by the combination of all the timing and resource constraints. These additional precedences refine the distance matrix (Section 2.2.2), thus providing a more accurate estimate of the set of feasible start times. It will guide the decisions made in the scheduler and often prevent it from making decisions leading to infeasibility.

After constraint analysis, the worst case or upper bound *ub* is computed for all the values being assigned into register file RF. It corresponds to the requirement

of registers in the worst case when scheduling is roughly performed without noticing the register file constraints. The upper bound is obtained by exact coloring the worst-case conflict graph (WCCG) (Section 6.4). If it is larger than $C(RF)$, some potential conflicts violating the register file capacity constraint have to be solved by reducing the schedule freedom.

Lower bound is used to deduce the feasibility of a schedule with the register file capacity constraints and address range constraints. A lower bound *lb* is determined by coloring the strong conflict graph SCG (Section 6.4) and is compared with the capacity $C(RF)$. If it is larger than the capacity of the register file RF, then an infeasible case is determined. Upper bound and lower bound give a general overview of the RF ability for a certain application, while the detailed register allocation has to be worked out.

In order to model that certain values can reside in a limited address range S_i of a register file, we propose the step *ACG analysis*. Assuming there are m ranges for RF. Each range with S_i registers can be viewed as a virtual register file VRF_i conceptually, with its capacity equal to the number of registers in S_i. Therefore we can write as follows:

$$C(VRF_i) = |S_i| \qquad \exists i = 1, \ldots, m \tag{6.1}$$

The capacity of RF equals to the number of registers in the union of S_i. We denote the union as S.

$$C(RF) = |\cup_{i=1}^{m} S_i| = |S| \tag{6.2}$$

Therefore, together with $C(RF)$, multiple $C(VRF_i)$s need to be satisfied. To avoid an extra phase of address range assignment, i.e. to assign each value to a virtual register file, register allocation is still performed on the uniform RF. Virtual register file constraints are translated into *label set* constraints in the conflict graph. We perform this by annotating each value with a label set which contains all the registers in which it can be assigned. By refining the conflict graph with all the label sets, the annotated conflict graph (ACG) therefore contains all the limited address range constraints. Thus it can be used for coloring just as conventional register allocation.

After ACG analysis, pairs of values that may potentially be stored in the same register file have to be identified in order to reduce the maximum number of conflicts, i.e. *bottleneck identification*. Since the pairs of values are potentially conflicting, i.e. they have weak conflicts, the bottleneck identification is performed on the WCCG. Several heuristics based on saturation number and degree number are discussed in [4] in order to select the most critical conflicts. This identification decision will cause values with weak conflicts to be allocated in the same register.

Lifetime serialization will try to meet the constraints by serializing the potential overlapping lifetimes. This is performed by adding sequence edges between the identified pairs of values [4]. Heuristics are also provided in sacrificing minimum distances in order to keep as much schedule freedom as possible. The constraint analysis calculates the effect of serialization on the schedule freedom of all the operations. This will make sure that decisions causing infeasibility are avoided.

The upper bound and lower bound calculation, ACG analysis, bottleneck identification, lifetime serialization and constraint analysis are performed alternatively until the capacity constraint matches the register file requirements.

6.4 Constructing a conflict graph

Often the assignment of an operation to a clock cycle (the "ultimate" scheduling decision) is delayed until all the resource and timing constraints are satisfied. This idea of lifetime conflict becomes unclear if the lifetimes are not fixed yet. In order to represent all the potential conflicts, three situations are classified in [4] as follows.

- strong conflict: values u and v have strong conflict if their lifetimes overlap for sure. There is overlap between u and v iff the production of value v is before the consumption of value u **and** the production of value u is before the consumption of value v.
- no conflict: values u and v have no conflict if their lifetimes can never overlap. There is no overlap between u and v iff the consumption of value v is before the production of value u **or** the consumption of value u is before the production of value v.
- weak conflict: values u and v have weak conflict if neither of the above conditions holds.

Two different conflict graphs are used in this approach, The first is the worst-case conflict graph (WCCG) which represents the worst case when all the potential conflicts become strong conflicts in the worst case. This graph contains all the weak conflicts and strong conflicts. Examples of WCCG are depicted in Figure 6.4 (b) for a DFG in Figure 6.4 (a). In Figure 6.4 (b), a solid edge represents a strong conflict, and a dashed edge represents a weak conflict. For example, values a and b have a strong conflict for sure since they are consumed by the same operation. Values a and e have a weak conflict because their relationship depends on the scheduling decisions.

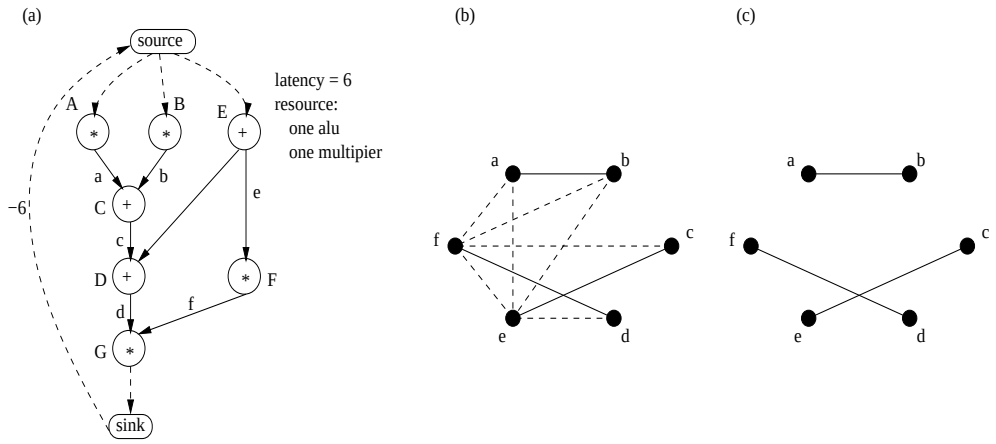


Figure 6.4: A DFG with its WCCG and SCG

Coloring the graph results the chromatic number of 4, while by accumulating all the registers in the existing register files we obtain the number of 2. Therefore some lifetime serialization has to be performed. The best case or strong conflict graph (SCG) represents the case when values have strong conflicts. This is depicted in Figure 6.4 (c) and its chromatic number is 2, which means at least two registers are needed for this application.

6.5 Annotated conflict graph analysis

The conflict graph constructed in the previous section has been used for conventional register allocation. In this section, it is modified with annotations and the annotated conflict graph (ACG) is proposed for register allocation with address range limitations.

6.5.1 Limitations on the conventional conflict graphs

Graph coloring is frequently used for register allocation in general-purpose and embedded processors. The general approach is to analyze the lifetimes of values in an application and assign values having a lifetime conflict with different colors, thus into different registers.

Using the exact graph coloring algorithm in [19], one color is assigned to each node in the conflict graph respecting to the lifetime conflicts. There is no restriction on which exact color to use until all the colors are used up for the

maximum clique. When a value is limited to certain address range, it implies that it can only be assigned with limited colors in the conflict graph. Therefore the original graph has to be adapted to these constraints.

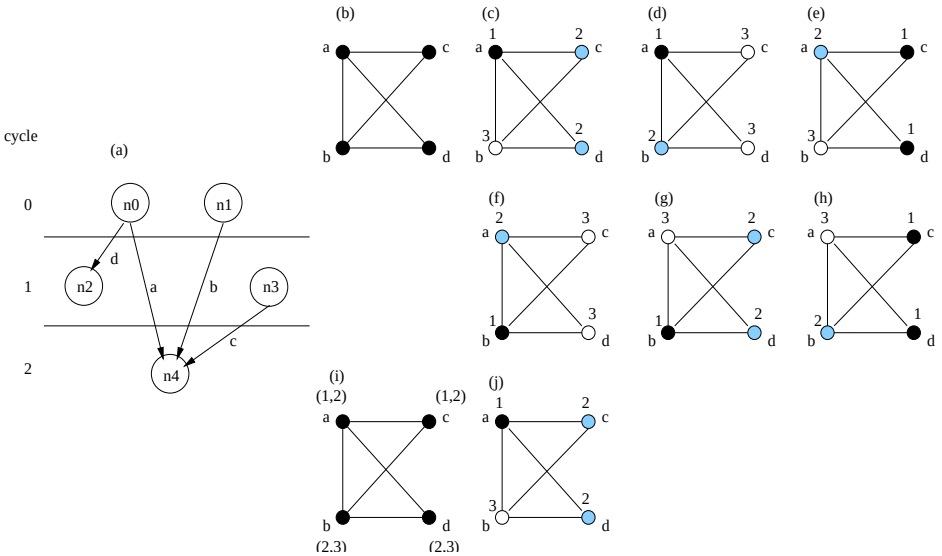


Figure 6.5: A scheduled DFG, the conflict graph and the labeled conflict graph

The following example illustrates this limitation. For simplicity, we assume that the scheduling has been performed and the scheduled DFG is presented in Figure 6.5 (a). The corresponding conflict graph is constructed in Figure 6.5 (b) according to the lifetime analysis. By coloring this conflict graph, we obtain that maximally three colors (registers) are needed for allocating all the values. We know, for example, value a cannot be put into the same register as value b, but there is no limitation on which concrete register value a can reside. All the possible allocation results are shown in Figure 6.5 (c), (d), (e), (f), (g) and (h). Assume that an LAR architecture is employed with three registers; further assume that values a and c can only be stored in register 1 or 2, and values b and d can only be stored in register 2 or 3. For each value we collect all the labels of the possible registers in which it can reside and assign a label set to the corresponding node. Therefore, label set $(1, 2)$ is associated with node a and c, and label set $(2, 3)$ is associated with node b and d. The annotated conflict graph (ACG) is constructed as in Figure 6.5 (i). For this simple example, we can see immediately that only one allocation result is possible and it is given in Figure 6.5 (j).

6.5.2 Limitations on the multiple register file approach

As we mentioned before, each address range is now modeled as a virtual register file, with the capacity equal to the number of registers in the range. The general problem in Section 6.3 is transferred to the scheduling and register allocation problem with multiple register files. Traditionally an additional register file assignment phase has to be performed for assigning a value to certain register file. In [5], graph coloring is used heavily for register file capacity satisfaction and register allocation. Previous work [4] extended this approach to multiple register files, in which the author assumes that register file assignment is performed in advance. However, it is either hard to make this decision during scheduling or making assignment decisions in early stage will cause non-optimal scheduling.

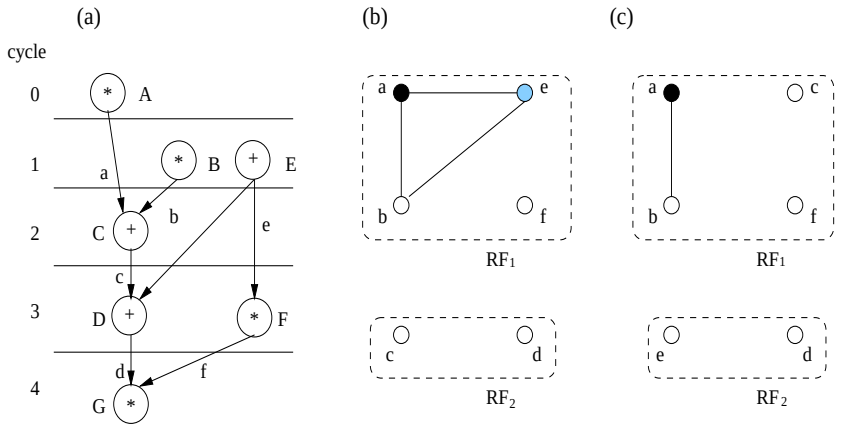


Figure 6.6: A scheduled DFG and the conflict graphs for multiple register files

An example is illustrated in Figure 6.6. Assuming values produced by functional unit MUL can only be stored in register file RF_2 and values produced by functional unit ALU can be stored in either RF_1 or RF_2. Also assume that $C(RF_1) = 2$ and $C(RF_2) = 1$. For the scheduled DFG in Figure 6.6 (a), if initial assignment makes the decision that value e is stored in RF_1 and value c is stored in RF_2, then by coloring the conflict graphs in Figure 6.6 (b), we obtain the result that RF_1 needs three registers and RF_2 needs one register; this is infeasible, since the capacity of RF_1 is exceeded. However, if initially e is assigned in RF_2 and c in RF_1, then by coloring the conflict graphs in Figure 6.6 (c) both register file capacities are satisfied and a feasible schedule is obtained.

This example shows that multiple register file assignment as a separate phase is obstacle to an optimal scheduling and register binding. The decision of which

register file to use has to be postponed and integrated with scheduling and register binding decisions.

6.5.3 ACG analysis

In the conflict graph, a node represents a value associated with a data edge in the DFG, and an edge between two nodes represents a lifetime conflict between the two values. In the labeled conflict graph, for a certain value v, its label set l_v includes all the k subsets of registers that it can be allocated in.

$$l_v = \cup_{i=1}^{k} S_i \tag{6.3}$$

The use of label set implies that it cannot be allocated into those registers which are not in this label set, or it has a *conflict* with those registers. The labeled conflict graph is enhanced by including all these conflicts and finally the ACG is constructed. Before doing this, we include a set of dummy nodes $\{s_j, j = 1, \ldots, |S|\}$ in the ACG, each one representing a register in the RF. There is a strong conflict between each pair of the dummy nodes since different registers are physically independent. In addition to those dummy nodes, there is a strong conflict between a node v and a register s_j if the label set l_v does not include the label of register s_j. Assuming that the set of strong conflicts in a conflict graph for a node v is represented as sc_v. We formalize annotated conflicts as follows.

$$sc_v = sc_v \cup sc_{v,s_j} \qquad if \qquad s_j \notin l_v \tag{6.4}$$

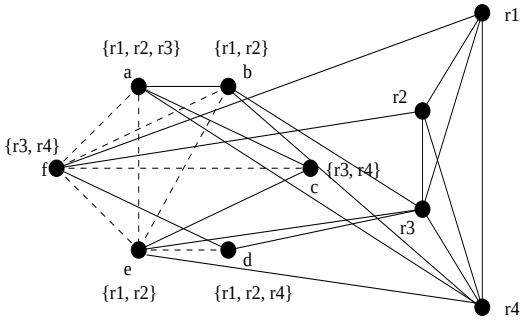


Figure 6.7: ACG analysis

The ACG for the DFG in Figure 6.4 (a) is presented in Figure 6.7. In this figure, four dummy nodes representing all the registers in RF are included in the ACG. They form a clique since each pair has a strong conflict. Assume that

register file RF is encoded into four subsets S_1, S_2, S_3 and S_4, which contains registers $\{r1, r2, r3\}$, $\{r1, r2\}$, $\{r3, r4\}$ and $\{r1, r2, r4\}$ respectively. Also assume that values a, b, c, d, e and f can be allocated in the subsets S_1, S_2, S_3, S_4, S_2 and S_3 separately, then the label set for each value can be included in the ACG. Moreover, the ACG is enriched by analyzing the conflicts between the nodes representing the values and the dummy nodes representing the registers according to equation 6.4. For example, there is a strong conflict between a and $r4$ since S_1 does not contain register $r4$. Subsequently coloring algorithms can be applied directly to the final ACG. This fits in well the trajectory of our global approach, since extra register file assignment phase is relieved and a uniform space for conflicts is maintained.

6.6 Experimental results

This section provides several experimental results by applying the SRM methodology to the ACG architecture. Based on the SRM concept, multiple ranges of a register file are modeled as virtual register files and alternative assignment choices for different address range are transferred as conflicts during graph coloring. Section 6.6.1 presents the results on limited read address range (LRAR) architecture with local reading and global writing, and compares them with the clustered architecture. Section 6.6.2 gives the result on different LAR architectures and shows the reduction on instruction encoding. Architecture design exploration is discussed in 6.6.3.

6.6.1 LRAR architecture

In this section, we show several experiments on a specific kind of LAR architecture, the so-called LRAR architecture, which is depicted in Figure 6.8.

In this architecture, reading values is limited to local registers as in clustered architectures, while writing is not limited in such a way that values can be written into any of the existing registers. The results of a two-range LRAR architecture are compared to that of a two-way clustered architecture. Architectures are defined by several parameters, i.e., $|\mathrm{FU}|$ is the number of functional units per type. We assume that functional unit types are ALU and multiplier. The DFG is constrained by latency L and initiation interval II. In Table 6.1, column 2 gives the minimum number of registers in each register file for the clustered architecture and column 5 gives the minimum number of registers in each range for the LRAR architecture. Column 3 and 6 report the execution time and column 4 and 7 show the mobility reduction. The mobility in a DFG is defined as the average difference between the ALAP and the ASAP start times of operations.

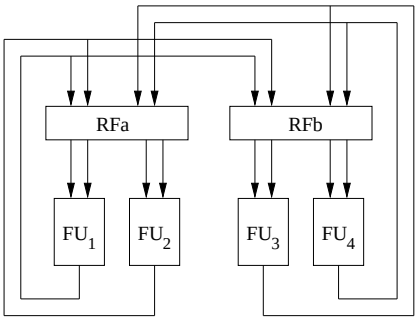


Figure 6.8: LRAR architecture

Table 6.1: LRAR architecture vs. clustered architecture

$DFG_{\|FU\|,L,II}$	*clustered*			*LRAR*		
	C	*T(s)*	*mob*	*C*	*T(s)*	*mob*
$fdct_{1,36,-}$	7,5	0.34	25.19 → 3.60	7,4	0.96	25.19 → 3.50
$fdct_{2,20,-}$	7,6	1.18	12.38 → 1.81	6,5	0.95	12.38 → 2.10
$fdct_{4,11,-}$	6,4	0.08	2.90 → 0.21	4,4	0.36	2.90 → 0.43
$loef31_{1,28,-}$	10,6	0.56	14.36 → 3.75	6,6	1.75	14.36 → 1.64
$loef31_{2,15,-}$	9,6	0.54	6.11 → 0.71	7,5	1.06	6.10 → 0.82
$loef31_{4,10,-}$	12,6	0.18	2.43 → 0.63	6,6	0.68	2.43 → 0.29
$fft_{1,13,-}$	4,1	0.09	3.17 → 0.73	4,1	0.06	3.17 → 1.77
$fft_{1,13,4}$	4,6	0.24	2.30 → 0.00	5,4	6.23	2.30 → 0.00
$fft_{2,11,-}$	4,2	0.08	2.17 → 1.23	4,2	0.08	1.27 → 1.70
$fft_{2,11,3}$	6,6	0.22	1.20 → 0.23	6,6	2.41	2.30 → 0.03
$ifft_{1,36,-}$	5,4	2.43	13.87 → 1.25	5,5	0.78	13.87 → 5.71
$ifft_{1,36,26}$	5,5	4.06	13.90 → 1.23	4,4	2.32	13.88 → 1.26
$ifft_{2,23,-}$	4,4	3.40	6.34 → 0.82	5,4	0.55	6.34 → 1.58

The results show that by modifying the architecture slightly and only allowing a wider address range for writing, the total number of registers can be reduced in general. This is because in the LRAR architecture, alternative choices exist for writing a value. If writing a value exceeds a particular range from where the value is read, it can be refined such that the value is written into another range. Thus the two ranges can be balanced and the total number of registers may be reduced. Notice that in our implementation the decision is made within a uniform search space instead of a step-wised refinement. In general the mobilities before

serialization for each benchmark are the same for the two architectures, except for $loef31_{2,15,-}$, $fft_{2,11,3}$ and $ifft_{1,36,26}$. This is because during implementation we added some dummy nodes for constructing the ACG, which changed the original DFG slightly. In some cases the register requirement is increased instead, such as in $ifft_{1,36,-}$. It can be explained as follows: since we use the SRM approach, all the values are bound into one big register file, which results in a large graph for coloring. Because heuristics are used for graph coloring and serialization, it is possible that serialization will select the weak conflict which is not a bottleneck and consequently results in a larger register file requirement.

6.6.2 Encoding reduction in LAR architecture

For an LAR architecture, the encoding can be reduced compared to a central register file architecture. This is because only necessary bits are used for the operand field. We perform the experiments on several benchmarks and compare the encoding requirements between a two-range LAR architecture and a central register file architecture. In Table 6.2, column 2 and 3 report the number of registers and encoding bits for the central register file architecture. Column 6 and 7 report the number of registers in each limited address range S_1 and S_2. Column 4 reports the total number of registers needed and column 5 reports the number of overlapping registers. Column 8 and 9 shows the encoding for the LAR architecture and in percentage compared to the central register file architecture.

For most of the benchmarks the code size is reduced and the maximum reduction is 17.65%. For $loef31_{2,15}$, even if the total number of registers is increased by one, the total encoding cost is still reduced by 17.65%. For some other benchmarks, e.g. $loef31_{1,28}$, the LAR architecture doesn't help in reducing the code size. The reason is that since there are a lot of swapping of values between the two ranges, the amount of operations that write to and read from different address ranges is large. Therefore the number of commonly addressable registers has to be kept large enough.

In order to improve the encoding reduction, one possible solution is to have more ranges and reduce the number of registers in each range. Table 6.3 shows the results for a four-range LAR architecture. In this table, we further divide each address range in the previous architecture as two subsets a and m. Moreover, a certain overlap is kept for these two ranges. In order to always guarantee a communication, each pair of the four ranges has overlapping.

As we can see, the code size is further reduced and the reduction increases from 17.65% to 21.01% for $fdct_{2,20}$ and from 17.65% to 23.82% for $idct_{2,15}$. This reduction is not obvious as we further partition the address ranges. The is because we always have to keep the number of overlapping registers large enough for communication. When $|S_i|$ is very small, the overlapping part almost covers

Table 6.2: Central register file vs. LAR architecture 1

$DFG_{\|FU\|,L}$	*central*		*LAR*					*%*
	C	*enc*	C				*enc*	
			$\|S\|$	$\|S_o\|$	$\|S_1\|$	$\|S_2\|$		
$fdct_{1,36}$	11	714	11	8	11	8	666	93.27
			11	9	11	9	714	100.00
			11	10	11	10	714	100.00
$fdct_{2,20}$	9	714	9	8	9	8	588	82.35
$fdct_{4,11}$	9	714	8	7	8	7	588	82.35
			8	6	8	6	588	82.35
$loef31_{1,28}$	12	952	12	9	10	11	952	100.00
			12	9	11	10	952	100.00
			12	8	10	10	952	100.00
			12	8	12	9	952	100.00
			13	8	11	10	952	100.00
$loef31_{2,15}$	11	952	12	4	8	8	784	82.35
			11	4	7	8	784	82.35
$idct_{2,15}$	9	680	9	5	8	6	560	82.35
			9	6	8	7	560	82.35
			9	7	8	8	560	82.35

Table 6.3: Central register file vs LAR architecture 2

$DFG_{\|FU\|,L}$	*central*		*LAR*							*%*
	C	*enc*	C						*enc*	
			$\|S\|$	$\|S_o\|$	$\|S_1\|$		$\|S_2\|$			
					a	m	a	m		
$fdct_{2,20}$	9	714	9	3	6	5	6	5	588	82.35
			9	3	6	4	6	5	564	78.99
$fdct_{4,11}$	9	714	9	2	5	4	6	4	588	82.35
$loef31_{2,15}$	11	952	12	4	8	6	8	6	784	82.35
			12	2	8	4	9	4	805	84.56
$loef31_{4,10}$	11	952	12	4	8	4	8	4	718	75.42
$idct_{2,15}$	9	680	10	3	7	5	6	4	536	78.82
			10	3	6	4	7	5	542	79.71
			10	3	7	4	6	4	518	76.18

the entire range, which will not improve the encoding reduction any more. From

Table 6.3, we can also see that in some cases, e.g. for $fdct_{2,20}$, even with the same total amount of registers, the address ranges can be tuned such that smaller code size can be obtained. Thus our approach can be applied to the design space exploration.

6.6.3 Design space exploration

Since the SRM concept is used for the LAR architecture, it can be easily applied to the design space exploration, as the methodology proposed in Chapter 5. The most obvious way is to keep the total number of registers unchanged and tune the address ranges, i.e. the capacity constraints for virtual register files. By performing scheduling and register binding, all the possible architectures satisfying timing and resource constraints can be obtained, from which the most efficient design can be selected. For reasons of convenience, we compare the experimental results between LRAR architecture and two-way clustered architecture. Table 6.4 and 6.5 show the different LRAR architectures for $fdct$ and $loef31$ benchmarks and Table 6.6 shows the loop folding results for *fft* and $ifft$ loop kernels.

Table 6.4: LRAR architectures for fdct benchmark

$DFG_{\|FU\|,L}$	*clustered*			*LRAR*			%
	C	*T(s)*	*mob*	*C*	*T(s)*	*mob*	
$fdct_{1,36}$	7,5	0.34	25.19 → 3.60	7,4	0.96	25.19 → 3.50	91.84
				6,5	0.95	25.19 → 3.50	100.00
				5,6	1.00	25.19 → 3.50	100.00
				4,7	0.95	25.19 → 3.50	86.73
$fdct_{2,20}$	7,6	1.18	12.38 → 1.81	6,5	0.95	12.38 → 2.10	100.00
				7,4	0.96	12.38 → 2.10	91.84
				8,3	0.95	12.38 → 2.10	91.84
$fdct_{4,11}$	6,4	0.08	2.90 → 0.21	4,4	0.36	2.90 → 0.43	85.56
	4,6	0.12	2.90 → 0.00	5,3	0.35	2.90 → 0.43	105.88

As we can see, for $fdct_{4,11}$ there are two possible clustered architectures which all satisfy the timing and resource constraints. For the other benchmarks, there is only one possible clustered architecture. On the contrary, there are multiple LRAR architectures with the same total number of registers which satisfy all the constraints. From them, the best design can be selected. For example, for $fdct_{1,36}$, four possible LRAR architectures are allowed. The last architecture obtains the smallest code size; therefore it can be selected as the best design.

Table 6.5: LRAR architectures for loef31 benchmark

$DFG_{\|FU\|,L}$	*clustered*			*LRAR*			%
	C	*T(s)*	*mob*	*C*	*T(s)*	*mob*	
$loef31_{1,28}$	10,6	0.56	14.36 → 3.75	6,6	1.75	14.36 → 1.64	87.77
				7,5	1.70	14.36 → 1.64	87.77
				8,4	1.71	14.36 → 1.64	81.19
				4,8	1.69	14.36 → 1.64	75.55
$loef31_{2,15}$	9,6	0.54	6.11 → 0.71	7,5	1.06	6.10 → 0.82	87.77
$loef31_{4,10}$	12,6	0.18	2.43 → 0.63	6,6	0.68	2.43 → 0.29	87.77
				7,5	0.67	2.43 → 0.29	87.77
				8,3	0.67	2.43 → 0.29	81.19

Table 6.6: Loop folding results for fft and ifft kernels

$DFG_{\|FU\|,L,II}$	*clustered*			*LRAR*			%
	C	*T(s)*	*mob*	*C*	*T(s)*	*mob*	
$fft_{1,13,4}$	4,6	0.24	2.30 → 0.00	5,4	6.23	2.30 → 0.00	100.00
				4,5	6.17	2.30 → 0.00	100.00
				3,6	6.18	2.30 → 0.00	100.00
$fft_{2,11,3}$	6,6	0.22	1.20 → 0.23	6,6	2.41	2.30 → 0.03	100.00
				7,5	2.40	2.30 → 0.03	100.00
				5,7	2.37	2.30 → 0.03	100.00
$ifft_{1,36,26}$	5,5	4.06	13.90 → 1.23	4,4	2.32	13.88 → 1.26	77.50
				5,3	2.33	13.88 → 1.26	91.43
				3,5	2.33	13.88 → 1.26	86.07

6.7 Conclusions and discussions

In this chapter, we propose a new type of architecture and encoding style for reducing the code size of embedded processors. The corresponding compilation techniques for register binding and scheduling for this architecture is also presented. The advantage is that by reducing addresses to a certain range of a register file, only necessary encoding bits are used in the instruction sets. In addition, we allow certain overlap between different ranges. Therefore swapping values between ranges are retained in the overlapping addresses and no extra hardware as well as no extra move operations are necessary. In order to support this architecture with efficient code generation, we apply the SRM concept to the architecture. The basic idea is that each range can be viewed as a virtual register file and the

capacity constraint has to be satisfied. To avoid an additional range assignment phase, register allocation is still performed on the central register file, while each value is associated with a label set. A label set contains all the possible registers in which the value can be assigned. Thus any non-existent assignment is transferred into conflicts when constructing the conflict graph. During register allocation, the address range constraints are automatically satisfied. The advantage of this approach is that scheduling and register allocation is performed on a uniform search space and it is blind to any range assignment decision. Another advantage is that it can be used easily for design space exploration, since only address ranges need to be modified. It can also be extended to model the network connectivities in the not-fully connected clustered architectures with several limitations.

One disadvantage is that the annotated conflict graph becomes very large for large benchmarks, which gives a heavy burden on the graph coloring of the worst-case conflict graph that is frequently used in the register allocation. Another disadvantage is that new instructions have to be introduced. The code size reduction is limited since the number of overlapping registers always has to be kept large enough. Once this number is not enough, extra move operations are still needed. In this thesis, we assume that source and destination operands always access the same range. In fact, architectures can be designed with more flexibility, such as source and destination operands access different overlapping ranges. Although our approach solved the phase coupling problem of address range assignment, we still assume that the operation assignment [13], i.e. the assignment of operations to functional units, is performed separately. In practice, it is desirable to combine the operation assignment with this approach.

Chapter 7

Conclusions and Future Work

This thesis presents a methodology for building a unified code generation of application specific embedded processors. Embedded processors are often required to be optimized not only for performance, but also for code size, area and cost. Applications written in high level languages are transferred into assembly language through processor instructions, and most of the constraints arise due to the encoding of instructions. Such constraints are very difficult to capture. In this thesis, we propose the static resource model (SRM) approach for modeling the constraints from those highly-encoded instruction sets and integrate them with the timing constraints from the signal processing application and resource constraints from the processor architecture. Constraint analysis techniques are the basis for administrating the search space with those integrated constraints. By focusing on the reduced search space, wrong decisions causing infeasibility are prevented. Similarly, it can also be applied to capturing the constraints from the limited address range (LAR) architectures. The SRM approach is used for the reconfigurable processor design since different instruction set configurations can be easily represented as static resources. It can be easily applied to design space exploration since constraints obtained through this modeling reflect directly the architecture specifications and are tunable by the designer.

The contribution of this work can be summarized as follows:

- A methodology of SRM is proposed for modeling the highly encoded instruction sets of application specific embedded processors. These constraints are combined with the resource constraints and timing constraints such that a unified code generation approach is obtained.

- Reconfigurable instruction set processors are looked at from a new angle. The reconfigurable part is moved from the data path to the instruction decoder. A new type of reconfigurable instruction decoder architecture tem-

plate is proposed, which can utilize the SRM concept directly and promise efficient code generation.

- The SRM approach also presents an efficient solution for instruction set design since all the constraints from instruction sets are represented as static resources, which can be use for fast estimation of performance. The instruction set can be tuned to obtain a better performance by modifying the static resources.

- This approach is also applied to the LAR register file architecture. In this architecture, the encoding of the operand field is limited to a certain range of a register file for the purpose of code size reduction. While it raises the problem of range assignment, the assignment decisions can be postponed by integrating the limited address range constraints with resource and timing constraints, resulting in a uniform search space.

The SRM is used heavily in this work for modeling the instruction set constraints and the constraints in the LAR architecture. The main advantages of this approach can be summarized as follows: in the first place, it is quite efficient with regards to the runtime of a compiler, since the SRM of a processor's instruction set needs to be computed only once and any resource-constrained scheduler can be applied without much adaptation. Secondly, the scheduler is not restricted by a specific code selection phase. Therefore it has more opportunity to minimize the register requirements. Thirdly, it can be used for fast estimation of performance of loop kernels. The scheduler also has the opportunity to produce optimal schedules for pipelined loops, which is very important for DSP applications.

The SRM approach can also be extended for modeling the connectivity constraints, such as buses and read/write ports. The disadvantage of this approach is that in case there are many irregularities in the architecture, the amount of virtual resources will become very large, which is an obstacle to the efficiency of the runtime of the compiler.

Another disadvantage is that it only captures the constraints on parallelism. It assumes there is no distinction between resources with the same type. For example, it assumes that functional resources which can execute the same type of operations are the same. In reality this is not always true. Although the LAR architecture can be used to capture the connectivity constraints to certain degree, it still doesn't solve the problem. Operations have to be assigned to functional resources with respect to connection constraints, as well as timing and resource constraints, and operation assignment has been studied in [13]. If operation assignment is performed separately from the instruction selection, then the phase coupling problem still exists. It is desirable that operation assignment can be integrated with this model while the efficiency of this approach is still preserved.

Future research includes:

- Investigate the integration of operation assignment with the SRM approach for code generation such that the connectivity constraints, instruction set constraints as well as timing and resource constraints can be integrated in the uniform search space.
- Modeling more versatile instruction set constraints, such as SIMD instructions in many processors, *superops* in Trimedia TM2 and some coarse grain instructions in reconfigurable processors.
- Improve the constraint analysis techniques such that it can deal with large amount of constraints efficiently.
- Automating the instruction set design and test for large instruction sets and for large amount of benchmarks.
- Investigate more versatile LAR architectures. In case that values can be assigned into more than one range, it is desirable that the address range assignment phase can be integrated with the current annotated conflict graph approach.
- In case functional unit read/write ports and register file read/write ports have to be included, the modeling becomes more complicated and study how much we can still gain from this approach.

For large applications containing conditional constructs and nested loops, basic blocks are too limited and the whole control-data flow graph has to be considered. Although if-conversion [88] is exploited to combine several basic blocks into one big block in case of conditional constructs, more elaborate approaches such as code motion need to be investigated. Moreover, a powerful machine description language needs to be exploited for a retargetable compilation for embedded processors.

Bibliography

[1] A.V. Aho. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, October 1989.

[2] A.V. Aho and S.C. Johnson. Optimal code generation for expression trees. *Journal of ACM*, 23(3):488–501, July 1976.

[3] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers–Principles, Techniques and Tools*. Addison-Wesley, MA, USA, 1986.

[4] C. Alba Pinto. *Storage Constraint Satisfaction for Embedded Processor Compilers*. PhD thesis, Eindhoven University of Technology, 2002.

[5] C. Alba Pinto, B. Mesman, and K. van Eijk. Register files constraint satisfaction during scheduling of DSP code. In *Proceedings of the XII Symposium on Integrated Circuits and Systems Design*, pages 74–77, Los Alamitos, CA, USA, October 1999. IEEE Computer Society Press.

[6] G. Araujo and S. Malik. Optimal code generation for embedded memory non-homogeneous register architectures. In *Proceedings of the 8th International Symposium of System Synthesis*, pages 36–41, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press.

[7] G. Araujo, S. Malik, and M. Lee. Using register-transfer paths in code generation for heterogeneous memory-register architectures. In *Proceedings of the 33rd Design Automation Conference*, pages 591–596, New York, NY, USA, 1996. ACM.

[8] A|RT designer, http://www.adelantetechnologies.com.

[9] P.M. Athanas and H.F. Silverman. Processor reconfiguration through instruction-set metamorphosis. *IEEE Computer*, 26(3):11–18, March 1993.

[10] D. Avis, D. Bremner, and R. Seidel. How good are convex hull algorithms? *Computational Geometry: Theory and Applications*, 7:265–301, June 1997.

[11] V. Bala and N. Rubin. Eficient instruction scheduling using finite-state automata. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 46–56, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press.

[12] S. Bashford and R. Leupers. Phase-coupled mapping of data flow graphs to irregular data paths. *Design Automation for Embedded Systems*, 4(2/3):119–165, 1999.

[13] M. Bekooij, B. Mesman, J.L. van Meerbergen, and J.A.G. Jess. Constraint analysis for operation assignment in FACTS. In *Proceedings of the 11st ProRISC/IEEE Benelux Workshop on Circuits, Systems and Signal Processing*, pages 229–236, Utrecht, The Netherlands, November 2000. STW Technology Found.

[14] N.G. Busa, A. van der Werf, and M. Bekooij. Scheduling coarse-grain operations for VLIW processors. In *Proceedings of the 12nd International Symposium on System Synthesis*, pages 47–53, Los Alatimos, CA, USA, 2000. IEEE Computer Society Press.

[15] A. Capitanio, N. Dutt, and A. Nicolau. Partitioned register files for VLIWs: A preliminary analysis of tradeoffs. *SIGMICRO-Newsletter*, 23(1-2):292–300, december 1992.

[16] Cdd package, ftp://ftp.ifor.math.ethz.ch/pub/fukuda/cdd/.

[17] B. Chazelle. An optimal convex hull algorithm in any fixed dimension. *Discrete Comput. Geom.*, 10:377–409, 1993.

[18] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, 1994.

[19] O. Coudert. Exact coloring for real-life graphs is easy. In *Proceedings of the 34th Design Automation Conference*, pages 121–126, New York, NY, USA, 1997. ACM.

[20] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[21] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Heidelberg, Germany, 1997.

[22] http://e-www.motorola.com/webapp/sps/.

[23] C. Eisenbeis, Z. Chamski, and E. Rohou. Flexible issue slot assignment for VLIW architectures. In *Conference Record the 4th International Workshop on Software and Compilers for Embedded Systems*, September 1999.

[24] A. Emmelmann, F.W. Schroer, and R. Landwehr. Beg-a generator for efficient back ends. *SIGPLAN Notices*, 24(7):227–237, July 1989.

[25] Embedded systems roadmap 2002, vision on technology for the future of progress, http://www.stw.nl/programmas/progress/index.html.

[26] J. Eyre. The digital signal processor derby. *IEEE Spectrum*, pages 62–68, June 2001.

[27] A. Fauth, J. van Praet, and M. Freericks. Describing instruction set processors using nML. In *Proceedings of the European Design and Test Conference*, pages 503–507, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press.

[28] M.M. Fernandes, J. Llosa, and N. Topham. Partitioned schedules for clustered VLIW architecture. In *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processors*, pages 130–134, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[29] A.W. Fraser. Engineering a simple efficient code-generator generator. *ACM Letter on Programming Language and Systems*, 1(3):213–226, September 1993.

[30] K. Fukuda. Frequently asked questions in polyhedral computation. Version 16 October 2000. http://www.ifor.math.ethz.ch/~fukuda/fukuda.html, 2000.

[31] M. Garey and D. Johnson. *Computers and Intractability. A Guid to the Theory of NP-Completeness*. W. Freeman, San Francisco, 1979.

[32] C.H. Gebotys. An efficient model for DSP code generation: Performance, code size, estimated energy. In *Proceedings of the 10th International Symposium on System Synthesis*, Los Alamitos, CA, USA, 1997. IEEE Computer Society Press.

[33] P. Grun, N. Dutt, and A. Nicolau. Memory aware compilation through accurate timing extraction. In *Proceedings of the 37th Design Automation Conference*, pages 316–320, New York, NY, USA, 2000. ACM.

[34] J.G. Gyllenhaal, W.W. Hwu, and B.R. Rau. Optimization of machine description for efficient use. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 349–358, Los Alamitos, CA, USA, December 1996. IEEE Computer Society Press.

[35] S. Hanono and S. Devadas. Instruction selection, resource allocation and scheduling in the AVIV retargetable code generation. In *Proceedings of the 34th Design Automation Conference*, pages 510–515, New York, NY, USA, 1997. ACM.

[36] R. Hartmann. Combined scheduling and data routing for programmable asic systems. In *Proceedings of European Conference on Design Automation*, pages 486–490, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[37] S. Hauck, T.W. Fry, M.M. Holser, and J.P. Rao. The Chimaera reconfigurable functional unit. In *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 87–96, Los Alamitos, CA, USA, April 1997. IEEE Computer Society Press.

[38] J.R. Hauzer and J. Wawrzynek. GARP: A MIPS processor with a reconfigurable coprocessor. In *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 12–21, Los Alamitos, CA, USA, April 1997. IEEE Computer Society Press.

[39] J. Hoogerbrugge and L. Augusteijn. Instruction scheudling for trimedia. *Journal of Instruction-Level Parallelism*, 1(1), February 1999.

[40] D.A. Huffman. A method for the construction of minimum redundancy codes. *Proceedings of the IEEE*, 40:1098–1101, 1952.

[41] J. Janssen and H. Corporaal. Partitioned register file for TTAs. In *Proceedings of the 28th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 303–312, Los Alamitos, CA, USA, November 1995. IEEE Computer Society Press.

[42] B. Kastrup, A. Bink, and J. Hoogerbrugge. ConCISe: A compiler-driven cpld-based instruction set accelerator. In *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 92–101, Los Alamitos, CA, USA, April 1999. IEEE Computer Society Press.

[43] D.C. Ku and G. De Micheli. *High-Level Synthesis of ASICs under Timing and Synchronization Constraints*. Kluwer Academic Publishers, Dordrecht, 1992.

[44] H. Lekatsas and W. Wolf. SAMC: A code compression algorithm for embedded processors. *IEEE Transactions on Computer-Aided-Design for Integrated Cirtuits and Systems*, pages 1689–1701, December 1999.

[45] R. Leupers. Register allocation for common subexpression in DSP data paths. In *Proceedings of the Asia and South Pacific Design Automation Conference with EDA TechnoFair*, pages 235–240, Piscataway, NJ, USA, January 2000. IEEE.

[46] R. Leupers and P. Marwedel. Algorithms for address assignment in DSP code generation. In *Proceedings of IEEE/ACM International Conference on Computer-Aided-Design*, pages 109–112, Los Alamitos, CA, USA, 1996. IEEE Computer Society Press.

[47] R. Leupers and P. Marwedel. Instruction selection for embedded DSPs with complex instructions. In *Proceedings of the European Design Automation Conference with EURO-VHDL*, pages 200–205, Los Alamitos, CA, USA, 1996. IEEE Computer Society Press.

[48] S. Liao, S. Devadas, K. Keutzer, and S. Tjiang an A. Wang. Storage assignment to decrease code size. *ACM Transactions on Programming Languages and Systems*, 18(3):235–253, May 1996.

[49] P.G. Lowney, A.M. Freudenberger, T.J. Karsez, W.D. Lichtenstein, R.P. Nix, and J.S. O'Donnell. The mmultiflow trace scheduling compiler. *Journal of Supercomputing*, 7(1-2):51–142, May 1993.

[50] MAJC architecture tutorual, http://www.sun.com/processors/majc.

[51] M.C. McFarland, A.C Parker, and R. Camposano. The high level synthesis of digital systems. *Proceedings of IEEE*, 78:301–318, February 1990.

[52] B. Mesman, M.T.J Strik, A.H. Timmer, J.L. van Meerbergen, and J.A.G Jess. A constraint driven approach to loop pipelining and register binding. In *Proceedings of Design Automation and Test in Europe*, pages 377–383, Los Alamitos, CA, USA, February 1998. IEEE Computer Society Press.

[53] B. Mesman, A.H. Timmer, J.L. van Meerbergen, and J.A.G. Jess. Constraint analysis for DSP code generation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(1):44–57, January 1999.

[54] B. Mesman, Q. Zhao, N. Busa, and K. Leijten-Nowak. Instruction set application tuning for dsp. *Journal for Circuits, Systems and Computers*, 12(3), June 2003.

[55] S. Novack, A. Nicolau, and N. Dutt. A unified code generation approach using mutation scheduling. In P. Mardwel and G. Goossens, editors, *Code Generation for Embedded Processors*. Kluwer Academic Publisher, Dordrecht, 1995. Chapter 12.

[56] D.B. Powell, E.A. Lee, and W.C. Newman. Direct synthesis of optimized DSP assembly code from signal flow block diagrams. In *Proceedings of International Conference on Acoustics, Speech and Signal Processing*, pages 553–556, New York, NY, USA, 1992. IEEE.

[57] R. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer, Heiderberg, Germany, 1985.

[58] T.A. Proebsting and C.W. Fraser. Detecting pipeline harzards quickly. In *Conference Record of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 280–286, New York, NY, USA, January 1994. ACM.

[59] S. Rathnam and G. Slavenburg. Processing the new world of interactive media. *IEEE Singla Processing Magzine*, 15(2):108–117, March 1998.

[60] B.R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. *International Journal of Parallel Programming*, 24(1):3–64, February 1996.

[61] B.R. Rau and C.D. Glaeser. Some scheduling techniques and an easily scheduable horizontal architechtre for high performance scientific computing. In *Proceedings of the 14th Annual Microprogramming Workshop*, pages 183–198, New York, NY, USA, 1981. IEEE.

[62] B.R. Rau, D.W.L. Yen, W. Yen, and R.A. Towle. The cydra 5 departmental supercomputer: Design philosophies and trade-offs. *Computer*, 22(1):12–35, January 1989.

[63] R. Razdan and M.D. Smith. A high-performance microarchitecture with hardware-programmable functional units. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 172–180, New York, NY, USA, November 1994. IEEE.

[64] K. Rimey and P.N. Hilfinger. Lazr data routinng and greedy scheduling for application-specific signal processors. In *Proceedings of the 21st Annual Workshop on Microprogramming and Microarchitecture*, pages 111–115, Washington, DC, USA, 1988. IEEE Computer Society Press.

[65] M.A.R. Saghir, P. Chow, and C. Lee. Exploiting dual-memory banks in digital signal processors. *SIGPLAN-Notices*, 31(9):234–243, September 1996.

[66] Sangiovanni-Vincentelli. A note on bipartite garphs and pivot selection in sparse matrices. *IEEE Transactions on Circuits and Systems*, 23(12):817–821, 1976.

[67] http://www.starcore-dsp.com/.

[68] A. Sudarsanam, S. Liao, and S. Devadas. Analysis and evaluation of address arithmetic capabilities in custom DSP architectures. In *Proceedings of the 34th Design Automation Conference*, pages 287–292, Ne York, NY, USA, 1997. ACM.

[69] A. Sudarsanam and S. Malik. Memory bank and register allocation in software synthesis of ASIPs. In *Proceedings of the IEEE/ACM Conference on Computer-Aided-Design*, pages 388–392, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press.

[70] Tensilica inc., http://www.tensilica.com/technology.html.

[71] TMS320C62xx CPU and instruction set: Reference guide.

[72] A.H. Timmer and J.A.G. Jess. Execution interval analysis under resource constraints. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 454–459, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[73] A.H. Timmer, M.T.J. Strik, J.L. van Meerbergen, and J.A.G. Jess. Conflict modelling and instruction scheduling in code generation for in-house DSP cores. In *Proceedings of the 32nd Design Automation Conference*, pages 593–598, New York, NY, USA, 1995. ACM.

[74] UTDSP benchmark suite, http://www.eecg.toronto.edu/ corinna/dsp/infrastructure/utdsp.html.

[75] C.A.J. van Eijk, B. Mesman, C.A. Alba Pinto, Q. Zhao, M. Bekooij, J.L. van Meerbergen, and J.A.G. Jess. Constraint analysis for code generation: Basic techniques and applications in FACTS. *ACM Transactions on Design Automation of Electronic Systems*, 5(4):774–793, October 2000.

[76] J.T.J van Eijndhoven and L. Stok. A data flow graph exchange standard. In *Proceedings of Europe Design Automation Conference*, pages 193–199, Los Alamitos, CA, USA, March 1992. IEEE Computer Society Press.

[77] J. van Praet, G. Goossens, D. Lanneer, and H. De Man. Instruction set definition and instruction selection for ASIPs. In *Proceedings of the International Symposium on System Synthesis*, pages 11–16, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[78] B. Wess. Automatic code generation for integrated digital signal processors. In *Proceedings of IEEE International Symposium on Circuits and Systems*, pages 33–36, New York, NY, USA, 1991. IEEE.

[79] B. Wess and M. Gotshlich. Optimal DSP memory layout generation as a quadratic assignment problem. In *Proceedings of the International Symposium on Circuits and Systems*, pages 1712–1715, New York, NY, USA, 1997. IEEE.

[80] T. Wilson, G. Grewal, B. Halley, and D. Banerji. An integrated approach to retargetable code generation. In *Proceedings of the 7th International Symposium on System Synthesis*, pages 11–16, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[81] M.J. Wirthlin and B.L. Hutchings. DISC: The dynamic instruction set computer. In *Proceedings of the International Society for Optical Engineering*, pages 92–103, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press.

[82] A. Wolfe and A. Chanin. Executing compressed programs on an embedded risc architecture. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 81–91, Los Alamitos, CA, USA, December 1992. IEEE Computer Society Press.

[83] R. Woudsma, R.A.M. Beltman, G. Postuma, A.C. Turley, W. Brauwer, U. Sauvagerd, B. Strassenburg, D. Wettstein, and R.K. Bertschmann. EPICS: A flexible approach to embedded DSP cores. In *Proceedings of the 14th International Conference on Signal Processing Application and Technology*, pages 506–511, Waltham, MA, USA, 1994. DSP Associates.

[84] Q. Zhao, T. Basten, and B. Mesman. Static resource models for instruction sets. In *Proceedings of the 14th International Symposium of System Synthesis*, pages 159–164, Los Alamitos, CA, USA, September 2001. IEEE Computer Society Press.

[85] Q. Zhao, B. Mesman, and T. Basten. Practical instruction set design and compiler retargetability using static resource models. In *Proceedings of Design Automation and Test in Europe*, pages 1021–1026, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[86] Q. Zhao, B. Mesman, and T. Basten. Static resource models for code-size efficient embedded processors. *ACM Transactions on Embedded Computing Systems*, 2(2):1–32, May 2003.

[87] Q. Zhao, B. Mesman, and H. Corporaal. Limited address range architecture for reducing code size in embedded processors. In *Proceedings of the 7th International Workshop on Software and Compilers for Embedded Systems*, Heidelberg, Germany, September 2003. Springer.

[88] Q. Zhao and C.A.J. van Eijk. Register binding for dsp code containing predicated execution. In *Proceedings of the 10th ProRISC/IEEE Benelux Workshop on Circuits, Systems and Signal Processing*, pages 611–618, Utrecht, The Netherlands, 1999. STW Technology Found.

[89] V. Zivojnovic, J. Martinez Velarde, and C. Schlager. Dspstone: A DSP-oriented benchmarking methodology. In *Proceedings of International Conference on Signal Processing Applications and Technology*, pages 715–720, Waltham, MA, USA, October 1994. DSP Associates.

Biography

Qin Zhao was born on April, 8, 1972 in Pinglang, Gansu province, China. She concluded her middle school in Pinglang No.1 middle school in 1989. From 1989 until 1996, she studied in the Department of Electrical Engineering, Southeast University, Nanjing, China, from where she got her B.Sc. degree in 1993 and M.Sc. degree in 1996 respectively. From April 1996 to December 1996, she worked as an lecture in Dongfei Display Research and Development Center, Southeast University. From January 1997 to December 1997, she was a trainee in Philips Display Components, the Netherlands. From January 1998, she worked towards a doctorate in the Information and Communication Systems Group, Department of Electrical Engineering, Eindhoven University of Technology, the Netherlands. She expected to receive this degree based on the work presented in this thesis on October, 23, 2003.

Since November, 2001, she has been a temporary researcher involved in the Ozone project in the Information and Communication Systems Group, Eindhoven University of Technology, the Netherlands.